PRIME Computer

PRIMOS SUBROUTINES REFERENCE GUIDE PDR3621

MAN 3251-001

PRIMOS SUBROUTINES
PDR3621
Revision A

This guide documents the operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 17 (Rev. 17).

PRIME

PRIME Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

ACKNOWLEDGEMENTS

We wish to thank the members of the documentation team and also the non-team members, both customer and Prime, who contributed to and reviewed this book.

First Printing January 1980

All correspondence on suggested changes to this document should be directed to:

Katherine S. Abrams
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

CONTENTS

PART I - OVERVIEW

PART II - PRIMOS SUBROUTINES

PART III MATH AND APPLICATION LIBRARY SUBROUTINES

PART IV INPUT/OUTPUT LIBRARY SUBROUTINES

PART VII - CONDITION MECHANISM SUBROUTINES

APPENDICES

Part I
Overview

SECTION 1

INTRODUCTION

This book describes the subroutines (including the operating system subroutines) that can be called from PRIME's high-level languages or the Prime Macro Assembler (PMA).

Procedures relating to building and modifying libraries and changing Input/Output Control System device assignments are included for user convenience. An overview of PRIMOS file system concepts and usage is in Section 3.

Libraries with subroutines that are useful for programmers are discussed in this guide. Other libraries, such as the COBOL (VCOBLB), RPG (RPGLIB), or PL/I (PL1GLB) libraries contain subroutines which are used exclusively by the appropriate compiler; the use of these libraries is discussed in the corresponding language user guide. (See Section 2 for a more detailed discussion).

DOCUMENT ORGANIZATION

This manual is divided into six parts which are described in the Table of Contents.

1. Overview
2. PRIMOS Subroutines
3. Math and Applications Library Subroutines
4. Input/Output Library Subroutines
5. Communication and Real-Time Library Subroutines
6. Library Management

CONVENTIONS

The following conventions are used in this guide.

Filename Conventions

filename	Source file
B_filename	Binary (object) file; compiler convention
L_filename	Listing file; compiler convention
M_filename	Map file
*filename	Saved executable memory image (R-mode)
#filename	Saved executable segmented runfile (V-mode)

C_filename Command file

PH_filename Phantom command input file.

O_filename Command output file

Filenames may be up to 32 characters long, the first character of which must be alphabetic (A-Z). Filenames can be composed only of the following characters: A-Z, Ø-9, _ # $ & - * . and /.

Note

On some devices, underline (_) may print as back arrow (<-).

Terminal Functions

(CR) or CR Carriage return

" Character erase; deletes last character in current line

? Line kill; deletes all characters in current line

^xxx Escape key for entry of non-printing character with ASCII code xxx

SECTION 2

INTRODUCTION TO SUBROUTINES

The subroutines described in this guide include PRIMOS System subroutines, Application Library subroutines and FORTRAN Mathematical subroutines. In addition to the standard FORTRAN math functions, Prime's library includes many other subroutines which can simplify high-level language programming. PMA programmers can make explicit use of the many low-level math and input/output subroutines that primarily support the language translators, but high-level language programmers will not normally need to call any of these low-level subroutines.

LOCATION OF LIBRARIES

The standard FORTRAN library subroutines for PRIMOS are contained in the files FTNLIB (R mode) and PFTNLB and IFTNLB (V mode) in UFD=LIB.

To get a list of all the libraries in the UFD LIB, use the commands:

```
ATTACH LIB
LISTF
```

To find the names of all the subroutines in any individual library, use the commands:

```
ATTACH LIB
EDB library
FIND ALL
QUIT
```

Shared FORTRAN, COBOL, FORMS, and MIDAS libraries may be installed at system startup time. Note that Rev. 15 or Rev. 16 shared libraries will not work with Rev. 17 PRIMOS and Rev. 17 shared libraries will not work with earlier versions of PRIMOS. For more information refer to the System Administrator's Guide.

A cross-reference of all subroutines described in this guide appears at the conclusion of the index.

The libraries described in this manual are:

Library	R mode	V mode
Applications	APPLIB	VAPPLB
Fortran and operating system	FTNLIB	PFTNLB IFTNLB
In-memory sorts	MSORTS	
Matrix	MATHLB	

Sort	SRTLIB	VSRTLI
Spool	SPOOL$	VSPOO$

There are other libraries not described in this manual. They are:

Library	R mode	V mode
Block device interface		BDVLIB
COBOL	COBLIB COBKID *	VCOBLB
FORMS	RFORMS	VFORMS
MIDAS	KIDALB	VKDALB
PL/I		PL1GLB
PRIMENET		VNETLB
RPG	RPGLIB RPGKID *	
Unimplemented Instruction Interrupt	UII	

* if MIDAS files are used

The subroutines in some of these libraries, such as PRIMENET, The Block Device Interface and MIDAS are discussed in other manuals. The calls to subroutines in other libraries, such as RPG, are generated automatically by compilers, etc. The details need never concern the programmer.

Note

At Rev. 17 of PRIMOS, the FORTRAN, MIDAS, COBOL and FORMS libraries and the UII package are assumed to be shared.

FILE-HANDLING SUBROUTINES

All file handling is done by a collection of special subroutines (Section 4), some internal to PRIMOS, and others available as library routines. These routines are used in common by PRIMOS and all Prime system software for simplified and uniform file handling. They can also be called from user programs. PRIMOS file handling subroutines are described in Section 4.

All the file handling subroutines called by the user are loaded when the FORTRAN library is loaded. Most of these subroutines are interlude subroutines which issue supervisor calls to PRIMOS in R-mode. Many file-handling subroutines are direct entrance calls to PRIMOS in V-mode. The appropriate subroutine in PRIMOS address space then executes the appropriate file operation.

File Handling in User Programs

The file-handling subroutines simplify communication between the PRIMOS file structure and user programs. In FORTRAN programs, for example, the symbolic device unit numbers in formatted READ and WRITE statements can be associated with PRIMOS file units. The following default assignments are set up by the compiler:

FORTRAN Unit (u)	File Unit (Funit)
5	1
6	2
7	3
8	4
9	5
10	6
11	7
12	8
13	9
14	10
15	11
16	12
17	13
18	14
19	15
20	16

Example: to write a record to file Unit 1 (FORTRAN Unit 5), the user could enter the command OPEN filename 1 2. The OPEN command associates the file Filename with the file unit 1 and opens the file for writing (code 2). During subsequent execution of a program containing a formatted WRITE statement such as:

```
WRITE (5, 10) LINE
```

the contents of array LINE are written to the FORTRAN Unit 5 (File Unit

1), according to FORMAT statement 10.

At the program level, a filename and funit number can be associated by the PRIMOS subroutine SRCH$$, as in:

```
CALL SRCH$$ (K$WRIT, 'TEXT', 4, 1, type, code)
```

See Section 4 for a more thorough discussion of SRCH$$.

File Input/Output: With the aid of the PRIMOS subroutine PRWF$$, the user can bypass formatted FORTRAN I/O and write directly from memory arrays to the file system, as in:

```
CALL PRWF$$ (K$READ, 1, LOC(text), 36, 000000, words, code)
```

This subroutine reads 36 words from the file associated with funit 1 to memory array text. words and code are returned values (words - number of words transferred, code - error code). 000000 is a 32-bit constant 0.

At the applications level, the Applications Library for file manipulation is also available for use.

INPUT OUTPUT (I/O) SUBROUTINES

The I/O subroutines are those relating to data transfers and device operations. The subroutines are managed by the Input/Output Control System (IOCS). The IOCS subroutines perform input/output between the Prime computer and the disks, terminals and peripheral devices within the system configuration. The I/O subroutines include:

- Device Independent Drivers which allow the user to maintain device independence by routing an I/O request to the independent driver (See Section 15).
- Device Dependent subroutines for non-data transferring functions required by I/O devices (See Section 16).
- File system subroutines which perform file system input/output operations (See Section 17).
- User Terminal subroutines which transfer data between a user terminal or ASR Reader/Punch and memory (See Section 18).
- Peripheral Device routines include routines that control line printers, drive a printer/plotter, drive serial and parallel card readers and drive 7-track and 9-track tapes (See Section 19).

FORTRAN LIBRARY

The FORTRAN Library File contains FORTRAN function subroutines and Math Subroutines.

- The FORTRAN function Library computational subroutines include the ANSI-standard functions. (See Section 7 for a description of these functions).
- Arithmetic subroutine calls are generated by the FORTRAN compiler when certain operations are specified in the FORTRAN program. These routines perform arithmetic operations on single precision integers, single and double-precision floating point and complex numbers. (See Section 9).
- Bit manipulation functions are provided by the FORTRAN compiler. In some cases the compiler will generate in-line code for these functions. However, in general, the library subroutine is needed. (See Section 8 for a description of these functions).

MATRIX LIBRARY

MATHLB (FORTRAN Matrix Subroutines)) contains subroutines to perform matrix operations, solve systems of simultaneous linear equations and generate permutations and combinations of elements. (See Section 10 for the scope and use of this library).

APPLICATIONS LIBRARY

The Application Library (APPLIB and VAPPLB) provides users with an easy-to-use library of service routines. They range from the very simple, which do little more that call a lower level routine, to those that are fairly complex because of the function desired. This library provides relatively high-level functions such as:

- String handling routines
- User query routines.
- System information routines.
- Mathematical routines
- Conversion routines
- File system routines.
- Parsing routines.

SORT LIBRARIES

There are three libraries containing sort subroutines:

- SRTLIB subroutines are used to perform file sorting operations.
- VSRTLI is the V-Mode version of SRTLIB.
- MSORTS library contains several in-memory sort subroutines and a binary search subroutine.

SYNCHRONOUS AND ASYNCHRONOUS CONTROLLERS

These subroutines perform the moving of raw data for assigned AMLC or SMLC lines. (Section 20).

REAL-TIME SUBROUTINES

PRIMOS supports user applications that have real-time requirements or the need to synchronize execution with other user programs. This support is a set of subroutines that provide access to Prime's semaphore primitives and to internal timing facilities. (See Section 21).

CALLING SEQUENCE CONVENTIONS

FORTRAN - Assembly Language Interface: The form of a call statement in FORTRAN is:

```
CALL name
CALL name (argument-1)
CALL name (argument-1, argument-2,...,argument-n)
```

where name is the subroutine name and argument-1,...,argument-n is a list of arguments. FORTRAN translates the CALL statement into a JST or PCL in the same way as the PMA CALL pseudo-operation. When arguments are specified,the compiler generates a pointer in the the same way as a PMA DAC statement for S-mode or R-mode code, or an AP statement for V-mode or I-mode code. Figure 2-1 illustrates three calling sequences for S-mode or R-mode: with no arguments, with one argument, and with three arguments. The associated code is also presented. Table 2-2 illustrates the corresponding calling sequences for V-mode or I-mode.

Main Program

No Arguments

```
CALL SUBX
```

One Argument

```
CALL SUBX
DAC  A
```

Two or More Arguments

```
CALL SUBX
DAC  A
DAC  B
DAC  C
 .
 .
 .
DAC  Ø
```

Subroutine

```
    ENT SUBX
    REL
     .
     .
     .
SUBX DAC **
     first instruction
     .
     .
     .
    JMP SUBX,*
```

```
    ENT SUBX
    REL
     .
     .
     .
SUBX DAC **
     CALL F$AT
     DEC 1
APTR DAC**
     first instr.
     .
     .
     .
     JMP SUBX,*
```

```
    ENT SUBX
    REL
     .
     .
     .
SUBX DAC **
     CALL F$AT
     DEC  n
APTR DAC **
BPTR DAC **
CPTR DAC **
     .
     .
     .
nPTR DAC **
     first instruction
     .
     .
     .
     JMP SUBX,*
```

Note: CALL SUBX is equivalent to EXT SUBX
JST SUBX

Figure 2-1. S-mode and R-mode Subroutine CALL Conventions

Main Program

No Arguments

```
CALL SUBX
```

One Argument

```
CALL SUBX
AP   A,SL
```

Two or More Arguments

```
CALL SUBX
AP   A,S
AP   B,S
AP   C,S
 .    .
 .    .
 .    .
AP   n,SL
```

Subroutine

```
     ENT   SUBX,SBX1
     SEG
SUBX first instruction
      .
      .
      .
     PRTN
      .
      .
      .
     LINK
SBX1 ECB  SUBX
```

```
     ENT   SUBX,SBX1
     SEG
SUBX ARGT
     first instruction
      .
      .
      .
     PRTN
      .
      .
      .

     DYNM APTR(3)
     LINK
SBX1 ECB SUBX,,APTR,1
```

```
     ENT   SUBX,SBX1
     SEG
SUBX ARGT
     first instruction
      .
      .
      .
     PRTN
      .
      .
      .

     DYNM APTR(3)
     DYNM BPTR(3)
     DYNM CPTR(3)
     DYNM DPTR(3)
      .     .
      .     .
      .     .
     DYNM nPTR(3)
     LINK
SBX1 ECB SUBX,,APTR,n
```

Note: CALL SUBX is equivalent to EXT SUBX
PCL SUBX

Figure 2-2. V-mode and I-mode Subroutine CALL Conventions

SECTION 3

FILE MANAGEMENT SYSTEM CONCEPTS

PURPOSE OF FILE SYSTEM

The purpose of the file system is to simplify the manipulation of large quantities of data using the computer. The major goals of the file system are:

1. Automatic (not manual) allocation of disk storage space for files
2. Referencing files by name
3. Clustering related information together

To accomplish the first goal, PRIMOS keeps a special file on each disk to record the available space on that disk. PRIMOS uses this information to allocate disk space automatically, and the average user need not concern himself with the allocation process, other than to know that it works.

Referencing files by name means selecting the desired file by giving the File Management System string of alphanumeric characters. The file system reserves one special file as a directory; it contains the names of other files and their locations on the disk. The system can find this Master File Directory (MFD) readily because both its name and its location are always the same.

The third goal is achieved in two ways. The first is to have many file directories; this allows like files to have their names and locations saved in one file directory. The second way is to allow nested file directories (i.e., a file directory may contain names not only of files, but also of other file directories.) Thus, each user may divide his files into appropriate groups and subgroups as he sees fit.

File directories also provide some degree of access protection to the files contained within them, because a password may be associated with each file directory. To examine the files in a directory, the user must first supply the password for that directory.

USING THE FILE SYSTEM

To access files, the user must be attached to some file directory. A file directory is a file that contains the names of other files on the disk and the location on the disk of these files. A file directory may contain the names of other file directories. To access files stored in a directory, the user must give the password for that directory. A

user is properly attached when the file system has been supplied with the proper file directory name and password, and it has found and saved the name and location of the file directory. It can therefore find and operate on all files contained in that file directory.

File Operations

The major operations on files are: initialization for access (open); access; shutdown and resource deallocations (close); and deletion.

File Units

A disk file which is opened for reading and/or writing has a set of associated pointers and status indicators. They comprise a file unit, and serve as an access port for the exchange of data between the disk file and the active program. One file at a time can be assigned to each unit. The files may be open on several different logical disk units at once. There are 128 file units available per user (16 under PRIMOS III, 15 under PRIMOS II). Units 1 thru 126 may be used for any purpose. Unit Ø is reserved for the system and unit 127 is reserved for the COMOUTPUT File.

Opening a File

A file may be opened for reading only, for writing only, or for both reading and writing. If a file is opened for reading only, it may be read, but it cannot be changed.

The operation of opening a file does the following :

1. Searches the file directory to see if the filename requested is there

2. Sets up tables and initializes buffers in the operating system

3. Defines a pseudonym for the file. This pseudonym is called the file unit number, and is the only name used for transfer of data to and from the file.

If a file is opened for writing only, or for reading and writing, it may be changed; if the filename is not found in the directory, the filename is added to the file directory, and a new file is created. When a new file is created at the time of opening, no information is contained in the file.

Using an Open File

Once a file has been opened, a file pointer is associated with the file. The file pointer indicates the next binary word to be accessed.

To understand how the file pointer works, imagine that the words in a file are serially numbered from Ø. The file pointer is then the number of the next word to be accessed in a file.

Use of the Open and Close Commands

Various ways are provided to associate a specific filename (Filename) with a PRIMOS file unit number. One method is the OPEN command. Example:

```
OPEN filename funit key
```

Where filename is the name of a file listed in the UFD to which the user is currently attached; funit is a PRIMOS file unit number (1-126), and key is 1 for reading, 2 for writing, 3 for reading and writing, etc.

From the terminal, the user can open files with the OPEN command, and can close them with the CLOSE command. The OPEN command allows a user to assign a file to a unit and specify the activity - reading, writing, or both. For complete descriptions of commands, refer to the PRIMOS Commands Reference Guide (FDR31Ø8). File units 1 to 126 (1-15 under PRIMOS II) may be specified by the user.

Unit 16 is reserved for system use under PRIMOS II.

When the user is communicating with the file structure through one of the standard Prime translator or utility programs, files are referred to by name only. PRIMOS, or the program itself, handles the details of opening or closing files and assigning file units. For example, the user can enter an external command such as ED FILE1, which loads and starts the text editor and takes care of the details of assigning the file FILE1 to an available unit for reading or writing.

Because open-for-write files are subject to alteration (deliberate or accidental), the user must keep files closed except when they are being accessed. Open files absorb system resources and may also make these opened files unavailable to other users. The CLOSE ALL command returns all open file units to a closed and initialized state (except the command output file). When control returns to PRIMOS via an error condition, files are not closed.

On an open file, information may be read from the file starting at the file pointer into high-speed memory, or information may be written to the file starting at the file pointer.

Access and File Pointer

When a file is accessed, the file pointer is incremented once for each binary word accessed.

Positioning a File

The file pointer may also be moved backward and forward within a file without moving any data. This is called positioning a file. The value of a file pointer is called the position of the file. Positioning a file to its beginning is often called rewinding a file.

Truncation of a File

It is possible to shorten a file by truncating it. When a file is truncated, the part of the file that is located at or beyond the file pointer is eliminated from the file. If the file pointer is positioned at the beginning of the file, all of the information in the file is removed but the filename remains in the file directory.

Closing a File

A file that has been opened may be closed. The file unit number (pseudonym) and the corresponding table areas in the operating system are "cleaned up" and released for reuse.

Deleting a File

A deleted file has its filename removed from the file directory, and all of the disk memory that the file occupied is released for use by other files.

Write-Protected Disks

Using the file management system, it is possible to run with WRITE-PROTECTED disks.

FILE TYPES

A disk storage medium is composed of many separate blocks of data recording space (disk records or sectors). How these blocks are put together to make a file can greatly affect the efficiency of positioning. Because of this, the file system has two different ways of linking physical disk records together to form a file. The SAM (Sequential Access Method), results in more compact storage on the disk and requires less high-speed memory for efficient operation, but is much slower for repeated random positioning over a file. The DAM (Direct Access Method), results in quicker positioning over a file, but requires more disk space and more high-speed memory. SAM and DAM files are functionally equivalent in all other respects. The structural differences between these two file types are transparent to the user.

SAM Files

A SAM file is the basic way of structuring disk records into an ordered set; (i.e., a threaded list of physical disk records.) See Figure 3-1.

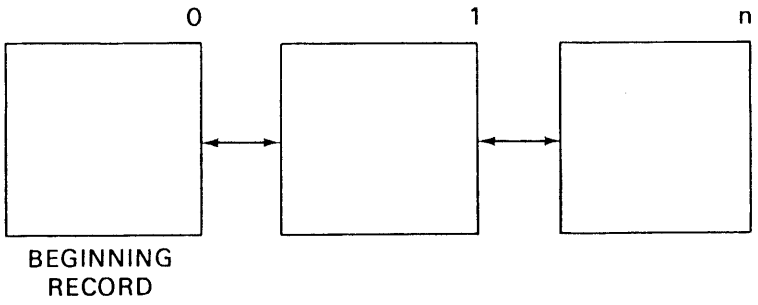


Figure 3-1. SAM File Structure

A SAM file is a collection of disk records chained together by forward and backward pointers to and from each record (See Appendix E). Each record in a SAM file (or any file) contains a pointer to the beginning record address (BRA) of the file. The first record has a pointer to the directory in which this file is an entry (father pointer). The file system maintains the record headers and is responsible for the structure of the records on the disk.

DAM Files

DAM (direct access method) file organization uses the SAM file method of making an ordered set; a special technique is used to rapidly access the i'th data record.

1. Logical file record Ø of a DAM file is reserved for use by the system. No user data is ever written in this record which is always the top level index.

2. The top level index is always one record long (exactly). If the file is short, the record address pointers point to records containing user data. Otherwise, the pointers point to records containing a lower level index. See Figure 3-2.

A DAM file index can exceed 512 entries on a storage module (220 entries for other devices). A multi-level index is maintained so that any record in the file can be directly accessed. (See Section 6 for DAM file creation example).

Figure 3-3 shows a typical relationship of DAM files within the PRIMOS file hierarchy.

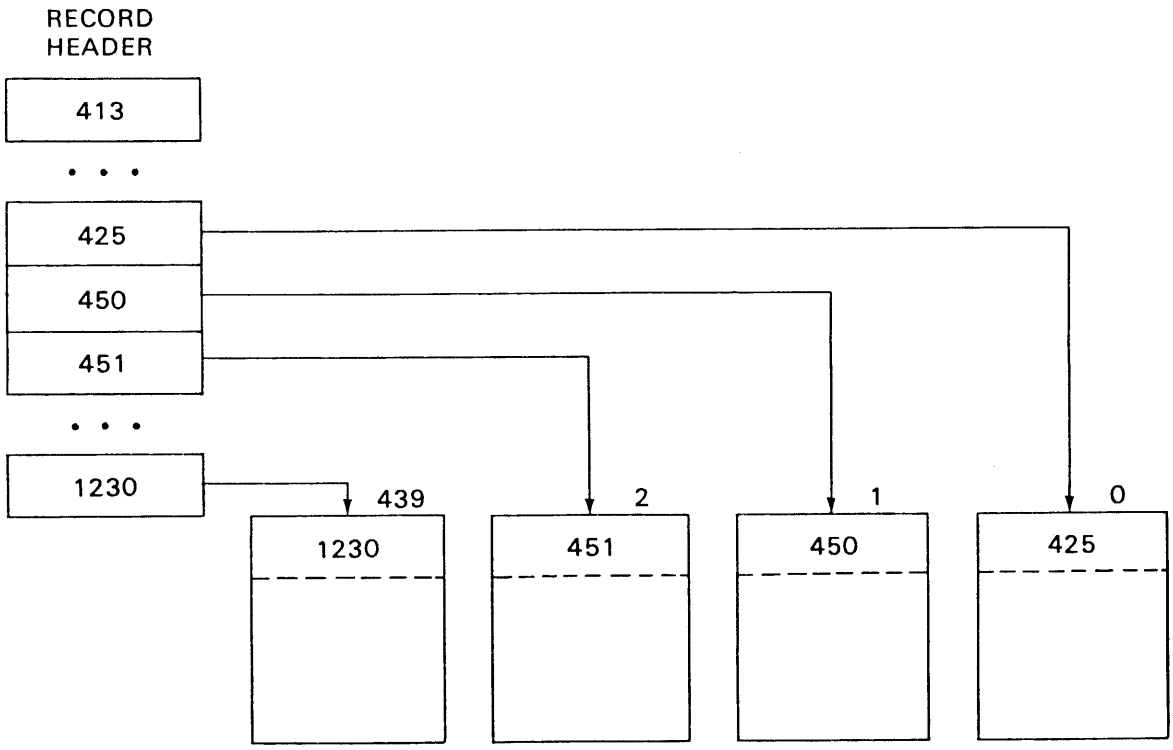


Figure 3-2 DAM File Structure

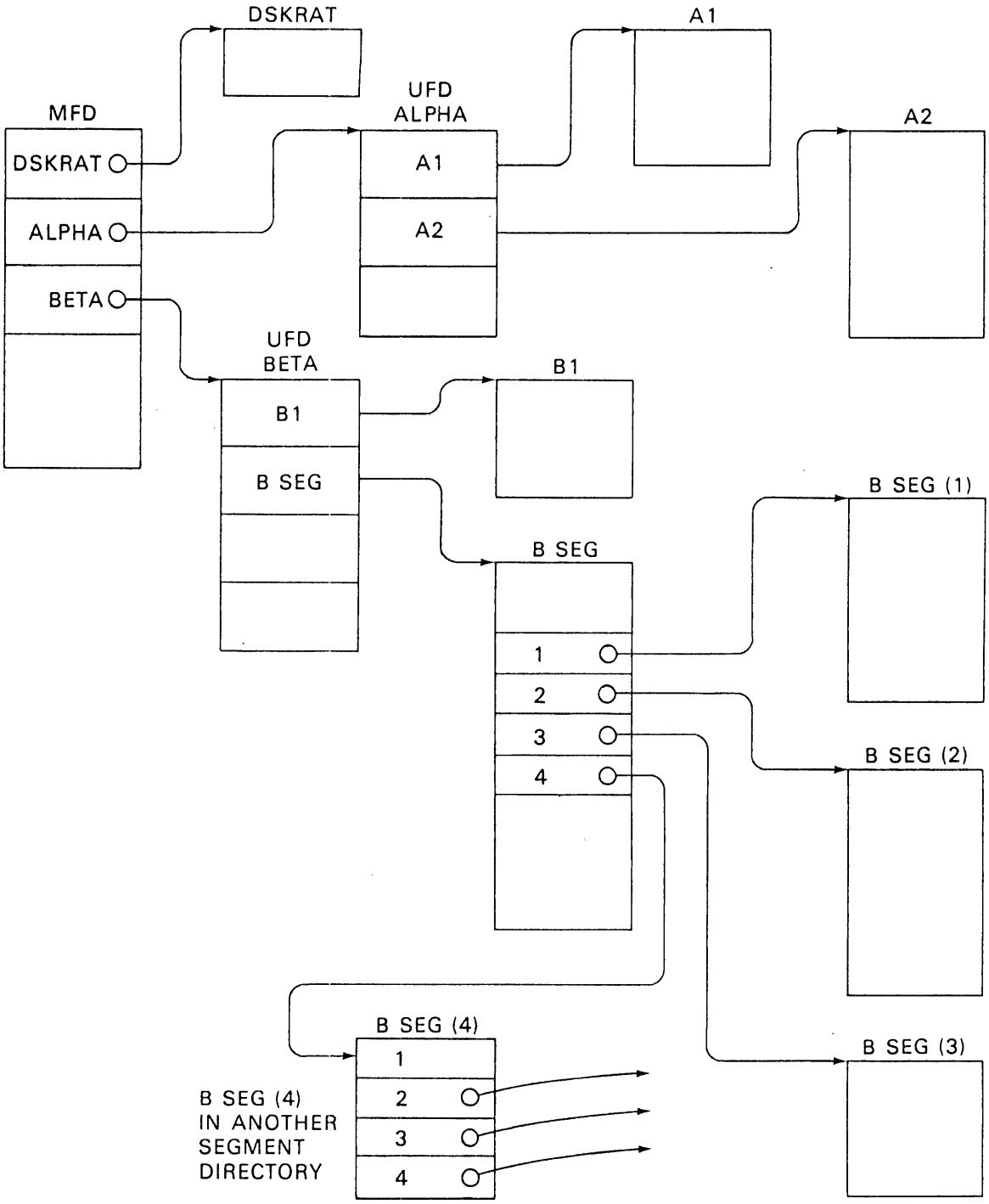


Figure 3-3. Hypothetical PRIMOS File Hierarchy with SAM and DAM File Structures

Record Formats

All files on PRIMOS disks are stored in fixed-length 448-word records, (1040-word records for storage module disks), chained together by forward and backward pointers. The number of records in a file is limited only by physical storage space.

The first eight words of the record make up the record header (first 16 words for storage module record). Specific content of record headers is discussed in Appendix E. All remaining words within the record, following the record header, may be used to store ASCII character pairs or 16-bit words. For further information about disks and storage modules, refer to the the System Administrator's Guide.

File Formats

A file is a series of records of the type described above, with the distinction that the first record in such a chain is reached from a pointer within a User File Directory or an entry in a segment directory.

Every file contains a series of 16-bit words. The format depends on the type of data in the file and how they were originally entered into the file system. The following types of files are in general use in PRIMOS systems:

File	Description
ASCII uncompressed	ASCII character text, packed two characters per word, as entered from a terminal or from the Prime card reader, paper-tape reader, etc. Each record is followed by a word containing a new-line character. This is the format of Source files, text and data records for sequential access.
ASCII Compressed	Same as above, but successive spaces are replaced by a relative horizontal tab character followed by a space count, and lines are terminated by a LINE FEED character.
Object	Translation of a source file as generated by the Macro Assembler and FORTRAN compiler for processing by the linking loader.
Memory Image	Header block followed by a direct transcription of high-speed memory. These files are created by LOAD and applications programs to be used as Runfiles.

Directories (UFD and Segment)	See Appendix E for format details.

FILE DIRECTORIES

Directories are specialized files containing entries that point to files or other directories. Directories are the nodes in the file system tree structure hierarchy; files are the branches. Figure 3-3 illustrates this concept. Directories are either User File Directories (UFD's) or segment directories. Each disk pack (or device, in the case of non-removable media) has one special UFD called a Master File Directory (MFD) that contains an entry for each User File Directory (UFD) in the MFD. In turn, each UFD contains an entry for every file or directory file in that directory. UFDs and MFDs are accessed in the same way as other files.

Segment directories differ from UFD's in one fundamental respect: they contain file locations but not file names. As far as the file system is concerned, the files in a segment directory have no symbolic names. However the user may refer to files within a segment directory by their entry number, which is a decimal number enclosed in parentheses as:

(1)
(2)
(185)

All of the above are 'names' of files in segment directories.

Master File Directory (MFD)

Each disk unit contains one MFD file as an index to the first physical record of each UFD in the MFD. The MFD has the same format as any UFD. The first record of the MFD begins at physical record 1 of the disk. Figure 3-3 shows a chain of pointers extending from the MFD to UFD and segment directories, and to a DAM or SAM file.

User File Directory (UFD)

A User File Directory (UFD) is a file that links PRIMOS filenames to the physical record of a file.

A UFD is associated with each user, project, etc. The UFD header includes the two passwords for the UFD. After the header, the UFD contains an entry for every file or directory named by the user. Each entry includes a filename and 2 words (INTEGER*4) that

contains the address of the first physical record of the file (called the beginning record address or BRA).

(See Appendix E for UFD header and entry details.)

UFDs can span multiple records; there is no limit to the number of files in a UFD.

UFD entries include an identification of special files; i.e., files having unique use in the file system and not normally accessed by the user. These files are BOOT, DSKRAT, BADSPT, and MFD.

Segment Directory Use

The segment directory file is opened for reading/writing on a unit of the user's choice. The file directory segment is then positioned to the segment directory entry number containing the desired file.

A desired file may be opened, closed, deleted, or truncated by giving the file unit number of the segment directory file rather that the filename. Segment directories are organized as SAM files or DAM files, consistent with the file structure the user wishes to build.

Segment Directory Formatting

A segment directory is formatted in a manner similar to a UFD except that entries are identified by a single entry number (from Ø to 65535) which is the pointer to the beginning record of a file. Segment directories are therefore limited to 65536 ('2ØØØØØ) entries.

A UFD entry in a segment directory is illegal. The only file types allowed in a segment directory are SAM, DAM, and other segment directories. See Section 6 for an example of creating segment directories.

Date/Time Stamping

There is a field in a file's UFD entry that records the date and time when the file was last modified. This field is updated when a file is closed, and either of the following conditions exist:

- An old file has been opened for writing or reading and writing, and a write operation has been performed.

- A new file has been created.

Notes

The father UFD is updated whenever entries are changed added, or deleted in that UFD.

The use of "last modified" rather than "last used" allows the use of WRITE-PROTECTED disks.

DISK STRUCTURES

Disk Record Availability Table (DSKRAT)

PRIMOS maintains a file, whose name is the partition name (packname), containing the used/unused status of every physical record on the disk. The partition name is given when the disk is created by the MAKE command. For example, the name of the documentation disk is DOCUMN, and the name of the DSKRAT file for this disk is DOCUMN. Each record is represented by a single binary bit; a '1' means the record is available, and a 'Ø' means it is in use. On a typical PRIMOS disk, the DSKRAT file is allocated several contiguous records. The DSKRAT file is maintained as a file on the disk, starting at physical record 2. The format of DSKRAT is shown in Appendix E.

Disk Organization

PRIMOS supports all Prime disk options. Prime software provides facilities for keyed indexed direct access files. Multiple disks are organized so that every fixed disk and every removable disk or partition is a self-consistent volume with its own bootstrap, DSKRAT, and MFD. Logical record zero is cylinder zero, head zero, sector zero on all options.

FILE ACCESS

Attaching to a UFD

To access files or use PRIMOS utility functions, the user must be attached to a UFD. Typically, during program development, each user attaches to a UFD reserved for program files with the ATTACH command. For further information, refer to PRIMOS Commands Reference Guide. Within executable programs, the user can attach to other UFDs; for example, to access data. At the program level, this is accomplished by the subroutine ATCH$$ (see Section 4).

File Access Control

PRIMOS (including PRIMOS III) gives a user who attaches with owner password (owner) the ability to open file directories to other users with restricted rights to the owner's files. Specifically, the "owner" of a file directory can declare, on a per-file-basis, the access rights a "nonowner" has over each of the owner's files. These rights are separated into three categories:

- Read Access (includes Execute Access)
- Write Access (includes over-write and append)
- Delete/Truncate rights

The owner of a UFD can establish protection keys for any file in the UFD: the owner access rights and the nonowner access rights. The owner password is required to obtain owner privileges. The nonowner password (if any) is required to obtain nonowner privileges. The command:

PASSWD owner-password nonowner-password

replaces the existing passwords in the UFD with a new owner-password and a nonowner-password. This command must be given by the owner while attached to the UFD. A nonowner is returned a "NO RIGHT" error. The command:

PROTECT filename okey nkey

replaces the existing protection keys on filename in the current UFD with the owner (okey) and nonowner (nkey) protection keys. Valid numbers for these keys are:

Ø no Access allowed

1 read Access only

2 write Access only

3 read and Write Access

4 delete/Truncate only

5 delete/Truncate and Read

6 delete/Truncate and Write

7 all Access allowed (Read/Write/Delete/Truncate)

The owner can restrict access to a file by the protection mechanism, which can be useful in preventing accidental deletion or overwriting. A nonowner cannot give the PROTECT command and achieve desired results.

The command will return the message "NO RIGHT" and return to PRIMOS command level.

A user obtains owner status to a UFD by attaching to the UFD, giving its name and owner password in the ATTACH command. A user obtains nonowner status to a UFD by giving its name and nonowner password in the ATTACH command.

A user can find out his owner status through the LISTF command. LISTF types the name of the current UFD, its logical device and O, if the user is an owner, or N if the user is a nonowner. LISTF then types the names of all files in the current UFD. An owner can determine the protection keys on all files in the current UFD through use of the file utility, FUTIL.

Other Features of File Access

The owner/nonowner status is updated on every ATTACH command and separately maintained for the current UFD and home UFD.

A user's privileges to files under a segment directory are the same as his privileges with the segment directory.

The protection keys of a newly created file are:

owner has all rights (7)

nonowner has none (Ø)

The passwords of a newly created UFD are:

owner password is blank

nonowner password is zero (any password will match)

A nonowner cannot create a new file in a UFD, or successfully give the CNAME, PASSWD, or PROTECT commands and a nonowner cannot open his current UFD for reading or writing (see the ATCH$$ command, Section 4 for further details).

In the context of file access control, the MFD has all the features of a UFD. Therefore, an MFD can be assigned owner/nonowner passwords, and the UFDs subordinate to the MFD may have their access controlled by protection keys, via the PROTECT command. If file access is violated, the error message is: NO RIGHT

PRIMOS II File Access Control

The PRIMOS II operating system does not observe file access control over individual files, but it is compatible to a degree with PRIMOS III and PRIMOS. Under PRIMOS II, a user cannot obtain access to a UFD by

ATTACHing with the nonowner password. If the owner password has been given, the ATTACH is successful, but subsequent access to files in the directory is not checked. Files created under PRIMOS II are generated with the same protection keys as under PRIMOS III and PRIMOS and the passwords of a newly created UFD are the same.

File Data Access Methods

Under PRIMOS, the means of file access is the Sequential Access Method (SAM) or the Direct Access Method (DAM) which are discussed earlier in this section. With both methods, the file appears as a linear array of words indexed by a current position pointer. The user may read or write a number of words beginning at the pointer, which is advanced as the data are transferred. A file service call (PRWF$$) provides the ability to position the pointer anywhere within an open file. File data can be transferred anywhere in the addressing range. When a file is closed and re-opened, the pointer is automatically returned to the beginning of the file. The pointer can be controlled by both the FORTRAN REWIND statement and PRWF$$ positioning.

With the DAM method of access, the file also appears to be a linear array of words, but this method has faster access times in positioning commands. PRIMOS keeps an index described earlier in this section to allow fast random positioning. User calls to manipulate SAM and DAM files are identical.

PRIMOS-LEVEL USER INTERACTION

PRIMOS commands fall into two major categories: the internal commands (implemented by subroutines that are memory-resident as part of PRIMOS) and external commands (executed by programs saved as disk files in the command UFD, CMDNCØ).

Command Activity

On receiving a command at the system terminal, PRIMOS checks whether it is an internal command, and if so, executes it immediately. Otherwise, PRIMOS looks in the command directory of Logical Disk Unit Ø for a file of that name. If the file is found, PRIMOS RESUMEs the file (loads it into memory and starts execution). All files in the command directory are assumed to be SAVEd memory image files, ready for execution. Most are set up to return automatically to PRIMOS when their function is complete or errors occur. The command line that caused the execution of the saved program is retained and may be referenced by the program to obtain parameters, options, and filenames via the RDTK$$ subroutine. To add new external commands, the user simply files a memory image program (SAVEd file) under the command directory UFD (CMDNCØ). Memory image files may also be kept in other directories and executed by the RESUME command.

Command Files

As an alternative to entering commands one at a time at the terminal, the user can transfer control to a command file by the command: COMINPUT. This command switches command input control from the terminal to the specified file. All subsequent commands are read from the file. One can assign any unit for the COMINPUT file and command files may call other command files. For detailed information on the COMINPUT command, refer to the PRIMOS Commands Reference Guide (FDR31Ø8).

Command files are primarily useful for performing a complicated series of commands repeatedly, such as loading an extensive system. Command files are also useful in system building when many files must be assembled, concatenated, loaded, etc., (for example, generating library files).

File Maintenance (FIXRAT)

To give the user an efficient and thorough way to check the integrity of data on a PRIMOS disk, PRIMOS provides a file maintenance program, FIXRAT, in the command directory, CMDNCØ. When FIXRAT is invoked as an external command, it checks for self-consistency in the structure of pointers in every record, file, and directory on the disk. If there are breaks in the continuity of double-strung pointers, discrepancies between the DSKRAT file and the reconstructed Record Availability Table, or other error conditions, FIXRAT prints appropriate error messages. FIXRAT asks the user to specify whether or not to take certain steps to repair a damaged file structure on a particular logical disk unit. For details and examples, refer to the FIXRAT description in the System Administrator's Guide (PDR31Ø9).

Part II
PRIMOS Subroutines

Part II (Sections 4, 5 and 6) describes the PRIMOS subroutines: A complete description of parameters is given for each subroutine; followed by notes on usage, brief examples of calls, and notes on compatibility with old file system functions.

- In Section 4, file manipulation subroutines are described.
- Section 5 describes other calls to I/O control system subroutines.
- The sample programs in Section 6 illustrate the use of the subroutines.
- The real-time subroutines that set system-wide semaphores are described in Section 22 and the old file system calls (obsolete) are found in Appendix F.

SECTION 4

FILE MANIPULATION SUBROUTINES

INTRODUCTION

Key Definitions for File System Calls

All keys and error codes are specified in symbolic, rather than numeric, form. These symbolic names are defined as PARAMETERs (for FORTRAN programs) and EQUs (for PMA programs) in $INSERT files present in a UFD on the master disk named SYSCOM. The key definition files are named KEYS.F for FORTRAN and KEYS.P for PMA. The error definition files are ERRD.F and ERRD.P. The user is urged to use these symbolic names. For convenience in recognizing old file system keys, these files are listed in Appendix G.

Error Handling Conventions

There are two error handling schemes. One scheme, called the integer error return code scheme (described in Appendix G), handles file system and semaphore subroutines. The other, involving alternate returns, handles I/O subroutines. (See Section 14.)

Filenames

Filenames may be 1-32 characters in length, the first character of which must be alphabetic. Filenames can be composed only of the following characters: A-Z Ø-9 _ # $ & * - . and /. Filenames may not contain embedded blanks.

Direct-Entrance Calls to PRIMOS

PRIMOS supports direct-entrance calls to certain supervisory procedures. Using this mechanism, routines such as SRCH$$, PRWF$$, etc., can be invoked directly via a PCL instruction thereby circumventing the overhead associated with a SVC entry into PRIMOS. Direct-entrance calls are available only from V-mode programs and will be correctly set up by using the V-mode FTN library.

Direct-entrance calls are through ECBs (entry control blocks) that are contained in gate segment 5, of the supervisor. Invalid calls or other references to segment 5 will cause the error messages UNDEFINED GATE or ILLEGAL PAGE REF.

The PRIMOS routines that can be entered via direct call, described in this section, are:

ATCH$$
CNAM$$
COMI$$
COMO$$
CREA$$
FORCEW
GPAS$$
GPATH$
NAMEQ$
PRWF$$
RDEN$$
RDLIN$
REST$$
RESU$$
SATR$$
SAVE$$
SGDR$$
SPAS$$
SRCH$$
TSRC$$
UPDATE
WTLIN$

The PRIMOS I/O subroutines that can be entered via direct calls, described in Section 14, are:

D$INIT
RRECL
WRECL

The error-handling subroutines that can be entered via direct calls and are part of the error handling scheme via SVCs are (Section 14):

ERRSET
GETERR
PRERR

Wake-up and notify subroutines, useful for real-time programming and synchronization between processes described in Section 21 are:

SEM$DR
SEM$NF
SEM$TN
SEM$TS
SEM$WT
SLEEP$

Under R-mode memory images on PRIMOS II or PRIMOS III, all operating system subroutines use the SVC interface described in Appendix C.

SUBROUTINE DESCRIPTIONS

The File Manipulation Subroutines are described below in alphabetical order.

Caution

Do not omit any arguments in calls to the subroutines described in this section. Do not specify as Ø (or any constant) any arguments returned by the subroutines. Never specify the integer return code as Ø. Always check the error code to see if the subroutine call was successful. It is essential to refer to Appendix G which covers the error handling scheme for these subroutines.

▶ ATCH$$

ATCH$$ attaches to a UFD and, optionally, makes it the home UFD. In attaching to a directory, the subroutine ATCH$$ specifies where to look for the directory. ATCH$$ specifies a User File Directory (UFD) in the Master File Directory (MFD) on a particular logical disk, a sub-directory in the current UFD, or the home UFD as the target-directory of the ATCH$$ operation.

CALL ATCH$$ (ufdnam,namlen,ldisk,passwd,key,code)

ufdnam — The name of the UFD to be attached. If key=Ø and ufdnam is the key K$HOME, the home UFD, is attached. If the reference subkey is K$IMFD or K$ICUR, ufdnam is either a Hollerith expression or the name of a three-word array that specifies a ufdname to attach to.

namlen — The length in characters (1-32) of ufdnam. namlen may be greater than the length of ufdnam provided that ufdnam is padded with the appropriate number of blanks. If ufdnam=K$HOME, namlen is disregarded.

ldisk — The number of the logical disk to be searched for ufdnam when key=K$IMFD. The parameter, ldisk, must be a logical disk that is started up. Other values are:

- K$ALLD — Search all started-up logical devices in logical device order, and attach to the UFD in which ufdnam appears in the MFD of the lowest numbered logical device.
- K$CURR — Search the MFD of the disk currently attached.

passwd A three-word array containing one of the passwords of ufdnam. passwd can be specified as Ø if attaching to the home UFD. If the reference subkey is K$IMFD or K$ICUR, passwd is either a Hollerith expression (1 to 6 characters) or the name of a three-word array that specifies one of the passwords of ufdnam. If passwd is blank, it is specified as three words, each containing two blank characters.

key Composed of two subkeys that are combined additively, a REFERENCE subkey and a SETHOME subkey. The REFERENCE subkey values are as follows:

K$IMFD Attach to ufdnam in MFD on ldisk.

K$ICUR Attach to ufdnam in current UFD (ufdnam is a subdirectory).

The SETHOME subkey, K$SETH, may be added to the REFERENCE subkey, e.g., K$IMFD+K$SETH, which will set the current UFD to the home UFD after attaching. If the REFERENCE subkey is K$ICUR, or if ufdnam is Ø, ldisk is ignored, and ldisk is usually specified as Ø.

code An integer variable set to the return code.

To access files, the file system must be attached to some User File Directory (UFD). This implies that the file system has been supplied with the proper file directory name and either the owner or nonowner password, and the file system has found and saved the name and location of the file directory. After a successful attach, the name, location and owner/nonowner status of the UFD is referred to as the current UFD. As an option, this information may be copied to another place in the system, referred to as the home UFD. The ATCH$$ subroutine does not change the home UFD unless the user specifies a change in the subroutine call. The user gets owner status if he gives the owner password, or gets nonowner status if he gives the nonowner password. The owner of a file directory can declare, on a per-file basis, what access a nonowner has over the owner's files. The nonowner password may be given only under PRIMOS and PRIMOS III. (Refer to the description of the commands SPAS$$ and SATR$$ in this section for more information.)

A BAD PASSWD error condition does not return to the user's program. PRIMOS command level is entered, and the user is not attached to any UFD. Other errors leave the attach point unchanged.

Examples

1. Attach to home UFD:

   CALL ATCH$$ (K$HOME,Ø,Ø,Ø,Ø,CODE)

2. Attach to UFD named 'G.S.PATTON', password 'CHARGE' in current UFD:

   CALL ATCH$$ ('G.S.PATTON',1Ø,K$CURR,'CHARGE',K$ICUR,CODE)

▶ CNAM$$

CNAM$$ changes the name of a file in the current UFD.

CALL CNAM$$(oldnam,oldlen,newnam,newlen,code)

oldnam	The name of the file to be changed.
oldlen	The length in characters of oldnam.
newnam	The new name of the file.
newlen	The length in characters of newlen.
code	An integer variable set to the return code.

The user must be the owner of the UFD of the file to change the name. CNAM$$ does not change the last modified date-time of the file or any of the other attributes of the file. However, the last modified date-time of the UFD in which the file resides is changed. CNAM$$ may cause the position of the file in the UFD to change with respect to the other files if the new name is longer than the old name. It is illegal to change the name of the MFD, BOOT, BADSPT. A NO RIGHT error message is generated if this is attempted.

▶ COMI$$

COMI$$ switches the command input stream from the user's terminal to a command file, or from a command file to the terminal.

CALL COMI$$(filnam,namlen,funit,code)

filnam	The name of the command file to switch the command input stream. If filnam is TTY, the command stream is switched back to the terminal and funit is closed. If filnam is PAUSE, the command stream is switched to the

terminal but the file unit specified by funit is not closed. If filnam is CONTINUE, the command stream is switched to the file already open on funit. The values: -TTY, -PAUSE, and -CONTINUE cannot be used as option names.

namlen The length in characters (1-32) of filnam.

funit The file unit (1-126 or 1-15 under PRIMOS II) on which to open the command file specified by filnam. Normally, File Unit 6 is used.

code An integer variable set to the return code.

▶ COMO$$

COMO$$ switches terminal output to file or terminal.

CALL COMO$$(key,filnam,namlen,xx,code)

key A word of flags specifying the action to be taken:

:000001 Turn TTY output off.

:000002 Turn TTY output on.

:000004 Reserved.

:000010 Turn file output off.

:000020 Turn file output on.

:000040 Append to filnam if filnam is being opened; close filnam if turning file output off.

:000100 Truncate filnam if filnam is being opened.

filnam An array containing the name of the file to be opened or 0.

namlen The length in characters (1-32) of filnam or 0.

xx Reserved. Should be specified as 0.

code An integer return code from the file system.

Routing of the terminal output stream is modified as indicated by the key. If TTY output is turned off, all printing at the terminal is suppressed until TTY output is re-enabled or until a unit '77 error message is generated. If a filename is specified, any current command output file is first closed. The new file is opened for writing on the

command output unit '77, and all subsequent terminal output is sent to the file. TTY output continues unless explicitly suppressed. Unless the APPEND option bit is set, the current contents of the file is overwritten. The parameter can be omitted by specifying a pair of blanks or a length of Ø.

Error messages (from ERRRTN, ERRPR$) force TTY output on, but leave the command output file open (i.e., the error message will appear both on the terminal and in the file). Disk error messages force TTY output on and file output off for the supervisor user (the file is left open). Unrecovered disk errors will do likewise for the user to whom the disk is assigned.

▶ CREA$$

CREA$$ creates a new UFD (a subUFD) in the current UFD and initializes the new UFD entry.

CALL CREA$$ (filnam,namlen,opass,npass,code)

filnam — The name to be given the new UFD.

namlen — The length in characters (1-32) of filnam.

opass — A three-word array containing the owner password for the new UFD. If opass(1)=Ø, the owner password is set to blanks.

npass — A three-word array containing the nonowner password for the new UFD. If npass(1)=Ø the nonowner password is set to Ø's. Any password given to ATCH$$ matches a nonowner password of Ø's.

code — An integer variable to be set to the return code from CREA$$.

Passwords can be set such that the password cannot be entered from the keyboard (i.e., the directory is accessible only from a program). In any case, passwords can be, at most, six characters long. Passwords less than six characters must be padded with blanks for the remaining characters. Passwords are not restricted by filename conventions and may contain any characters or bit patterns. It is strongly recommended that passwords do not contain blanks, commas, or the characters = ! ' @ { } [ ] ( ) ; ^ < > or lower case characters. Passwords should not start with a digit. If passwords contain any of the above characters or begin with a digit, the passwords may not be given on a PRIMOS command line to the ATTACH command.

Since the subroutine SRCH$$ does not allow creation of a new UFD, CREA$$ must be used for this purpose. Under program control, CREA$$ allows the action of the PRIMOS CREATE command.

CREA$$ requires owner rights on the current UFD.

For example, to create new UFD with default passwords of blanks for owner and 3*Ø for nonowner:

```
CALL CREA$$ ('NEWUFD',6,Ø,Ø,CODE)
```

▶ FORCEW

CALL FORCEW (Ø, funit)

funit — A file unit number from 1 to 126 on which a file has been opened.

The FORCEW subroutine immediately writes to the disk all modified records of the file that is currently open on funit. Normally this action is not needed, since the system automatically updates all changed file system information to the disk at least once per minute. Under PRIMOS II, the FORCEW routine has no effect.

▶ GPAS$$

GPAS$$ returns the passwords of a SUBUFD in the current UFD.

CALL GPAS$$ (ufdnam,namlen,opass,npass,code)

ufdnam — The name of the UFD with passwords to be returned. ufdnam is searched for in the current UFD.

namlen — The length in characters (1-32) of ufdnam.

opass — A three-word array that is set to the owner password of ufdnam.

npass — A three-word array that is set to the nonowner password of ufdnam.

code — An integer variable set to the return code.

GPAS$$ requires owner rights to the current UFD.

For example, to read passwords of SUBUFD into PASS(6) array:

```
CALL GPAS$$ ('SUBUFD',6,PASS(1),PASS(4),CODE)
```

▶ GPATH$

GPATH$ obtains a fully qualified pathname for an open file unit, or for current or home attach points. GPATH$ operates in V-mode only.

CALL GPATH$ (key, funit, buffer, bufflen, pathlen, code)

key — An integer variable specifying pathname to be returned (INTEGER*2). Possible values are:

- K$UNIT — Pathname of file open on file unit specified by funit will be returned (K$UNIT = 1).
- K$CURA — Pathname of current attach point will be returned (K$CURA = 2).
- K$HOMA — Pathname of home attach point will be returned (K$HOMA = 3).

funit — Specifies file unit number if key is K$UNIT, otherwise ignored.

buffer — The buffer where the pathname is to be returned.

bufflen — Specifies maximum buffer length in characters. If the pathname exceeds bufflen characters, data in buffer is meaningless and a code of E$BFTS is returned.

pathlen — Specifies the length in characters of the pathname returned in buffer. Characters beyond pathlen in buffer contain no useful information.

code — A standard error code. Possible values are:

- 000000 — No errors.
- E$BKEY — A bad key was specified.
- E$BUNT — A bad unit number was specified in funit.
- E$UNOP — Unit specified in funit is closed and name cannot be returned.

E$NATT Not attached to any node (keys K$CURA,K$HOMA).

E$BFTS The buffer specified with character length bufflen is too small to contain full pathname. The buffer contains no valid data.

The following are examples of information returned as the result of using GPATH$. The lower-case names define what information the examples (in upper case) actually represent.

```
<disk_name>MFD
<SPOOLD>MFD

<disk_name>ufd name
<SPOOLD>SPOOLQ

<disk_name>ufd_name1>ufd_name2>file_name
<SALESD>WEST.COAST>YTD.1979>MARCH

<disk_name>ufd_name>segment directory name
<OPSYST>PR4.64>VPRMOS

<disk_name>ufd_name>segment_directory_name>entry_number>entry_number
<DBDISK>DICTIONARY>WORDS>22>68
```

▶ NAMEQ$

NAMEQ$ is a LOGICAL function that compares two filenames for equivalence.

```
LOG = NAMEQ$ (filnam1,namlen1,filnam2,namlen2)
```

filnam1 The first filename for comparison.

namlen1 The length in characters of filnam1.

filnam2 The second filename for comparison.

namlen2 The length in characters of filnam2.

NAMEQ$ performs a character-by-character compare of filnam1 and filnam2 for the length of namlen1 or namlen2, whichever is shorter. The trailing characters of the longer name (if the names are not the same length) must be all blank for equality. The names supplied must be valid filenames.

NAMEQ$ will work correctly on numeric fields only if namlen1=namlen2.

▶ PRWF$$

PRWF$$ reads, writes, positions, and truncates SAM or DAM files.

CALL PRWF$$ (rwkey+poskey+mode,funit,LOC(buffer),nw,pos,rnw,code)

rwkey — This subkey, which cannot be omitted, indicates the action to be taken. Possible values are:

- K$READ — Read nw words from funit into buffer.
- K$WRIT — Write nw words from buffer to funit.
- K$POSN — Set the current position to the 32-bit integer in pos.
- K$TRNC — Truncate the file open on funit at the current position.
- K$RPOS — Return the current position as a 32-bit integer word number in pos.

poskey — A subkey indicating the positioning to be performed. Possible values are: (If omitted, same as K$PRER)

- K$PRER — Move the file pointer of funit the number of words specified by pos relative to the current position before performing rwkey.
- K$POSR — Move the file pointer of funit the number of words specified by pos relative to the current position after performing rwkey.
- K$PREA — Move the file pointer of funit to the absolute position specified by pos before performing rwkey.
- K$POSA — Move the file pointer of funit to the absolute position specified by pos after performing rwkey.

mode — A subkey that may be omitted or used to transfer all or a convenient number of words. Possible values are: (If omitted, read/write nw)

K$CONV Read/write a convenient number of words (up to the number specified by the parameter nw).

K$FRCW Perform write to disk from buffer before executing next instruction in the program.

funit A file unit number (1 to 15 for PRIMOS II, 1-126 for PRIMOS) on which a file has been opened by a call to SRCH$$ or by a command. PRWF$$ actions are performed on this file unit.

LOC (buffer) The data buffer to be used for reading or writing. If buffer is not needed, it can be specified as INTL(Ø).

nw The number of words to be read or written (mode=Ø) or the maximum number of words to be transferred (mode=K$CONV). nw may be between Ø and 65535.

pos A 32-bit integer (INTEGER*4) specifying the relative or absolute positioning value depending on the value of poskey.

rnw A 16-bit unsigned integer set to the number of words actually transferred when rwkey=K$READ or K$WRIT. Other keys leave rnw unmodified. For the keys K$READ and K$WRIT, rnw must be specified.

code An INTEGER*2 variable to be set to the return code.

pos is always a 32-bit integer, not a <record-number, word-number> pair. All calls to PRWF$$ must specify pos even if no positioning is requested. An INTEGER*4 Ø can be generated by specifying ØØØØØØ or INTL(Ø) in FTN, ØL in PMA.

poskey is observed for all values of rwkey except K$RPOS, for which it is ignored (the file position is never changed).

If rwkey = K$POSN, nw and rnw are ignored, and no data are transferred.

A call to read or write nw words causes nw words to be transferred to or from the file, starting at the file pointer in the file. Following a call to transfer information, the file pointer is moved to the end of the data transferred in the file. Using poskey of K$PREA or K$POSA, the user may explicitly move the file pointer to pos before or after the data transfer operation. Using a poskey of K$PRER or K$POSR, the user may move the file pointer backward pos words from the current position, if pos is negative or forward pos words if pos if positive. Positioning takes place before or after the data transfer, depending on the key. If nw is Ø in any of the calls to PRWF$$, no data transfer takes place, and PRWF$$ performs a pointer position operation.

The mode subkey of PRWF$$ is most frequently used to transfer a specific number of words on a call to PRWF$$. In these cases, the mode is Ø and is normally omitted in PRWF$$ calls. In some cases, such as in a program to copy a file from one file directory to another, a buffer of a certain size is set aside in memory to hold information, and the file is transferred, one buffer-full at a time. In the latter case, the user doesn't care how many words are transferred at each call to PRWF$$, so long as the number of words is less than the size of the buffer set aside in memory.

Since the user would generally prefer to run a program as fast as possible, the K$CONV subkey is used to transfer nw words, or less in the call to PRWF$$. The number of words transferred is a number convenient to the system, and therefore speeds up program run time. The number of words actually transferred is set in rnw. For an example of PRWF$$ use in a program, refer to Section 6.

The subkey K$FRCW guarantees that PRWF$$ will not return until the disk record (s) involved are written to disk. The write to disk will be performed before executing the next instruction in the program. Since the K$FRCW defeats the disk buffering mechanism, it should be used with care as it increases the actual amount of disk I/O. It should only be used when it is necessary to know that data is physically on a disk (e.g., as when implementing error recovery schemes).

The programmer is responsible for ensuring that only one process (user) is involved in the PRWF$$ call concurrently. The file may be open for use by several processes. The forced write applies only to the data written by the process performing the operation. See an example of the use of the K$FRCW later in this section.

On a PRWF$$ BEGINNING OF FILE error or END OF FILE error, the parameter rnw is set to the number of words actually transferred.

On a DISK FULL error, the file pointer is set to the value it had at the beginning of the call to PRWF$$. The user may, therefore, delete another file and restart the program (by typing START after using the DELETE command). This feature does not work with PRIMOS II.

During the positioning operation of PRWF$$, PRIMOS maintains a file pointer for every open file. The file pointer is a two-word integer, because files may be longer than 65,536 words. When a file is opened by a call to SRCH$$, the file pointer is set in such a manner that the next word that is read is the first word of the file. The file pointer position is Ø, for the beginning of file. If the user calls PRWF$$ to read 490 words, and does no positioning at the end of the read operation, the file pointer is set to 490.

Note

In V-mode, PRWF$$ only transfers words into the same segment as buffer. An attempt to read across a segment boundary will cause a wrap-around instead and read into the beginning of the segment. This is also true of writing from the address space.

Examples

1. Read the next 79 words from the file open on unit 1:

```
CALL PRWF$$ (K$READ,1,LOC(BUFFER),79,000000,NMREAD,CODE)
```

2. Add 1024 words to the end of the file open on UNIT (10000000 is just a very large number to get to the end of the file):

```
CALL PRWF$$ (K$POSN+K$PREA,UNIT,LOC(0),0,10000000,NMW,CODE)
CALL PRWF$$ (K$WRIT,UNIT,LOC(BFR),1024,000000,NMW,CODE)
```

3. See what position is on File Unit 15 (INT4 is INTEGER*4):

```
CALL PRWF$$ (K$RPOS,15,LOC(0),0,INT4,0,CODE)
```

4. Truncate file 10 words beyond the position returned by the above call:

```
CALL PRWF$$ (K$TRNC+K$PREA,15,LOC(0),0,INT4+10,0,CODE)
```

5. Position the file open on unit number UNIT to the tenth word used in the file and the first 10 words of ARRAY will be written to it:

```
      INTEGER*2 ARRAY(40), CODE,UNIT,RET
$INSERT SYSCOM>KEYS.F
      CALL PRWF$$(K$WRIT+K$FRCW+K$PREA,UNIT,LOC(ARRAY),
     X            10,INTL(10),RET,CODE)
      IF (CODE .NE. 0) GOTO error_processor
```

The above FORTRAN call will cause the file open on unit number UNIT to be positioned to the tenth word in the file, and the first 10 words of ARRAY will be written to it. The next instruction in the user's program will not be executed until the data has actually been written to disk. If an error is encountered while writing to disk, the error code E$DISK (disk I/O error) is returned. If more than one concurrent user of the disk record is detected, the error code E$FIUS (file in use) is returned. In this case, the write is not lost, but will not be perfomed immediately.

RDEN$$

RDEN$$ positions in or reads from a UFD.

CALL RDEN$$ (key,funit,buffer,buflen,rnw,filnam,namlen,code)

key — An integer variable specifying the action to be taken. Possible values are:

- K$READ — Advance to the start of the first or next UFD entry and read as much of the entry as will fit into buffer. Set rnw to the number of words read.
- K$NAME — Position to the start of the entry specified by filnam and namlen. Read as much of the entry as will fit into buffer. Set rnw to the number of words read. If the entry is not in the directory, the code E$FNTF is returned. If namlen is zero, the next entry is returned.
- K$GPOS — Return the current position in the UFD as a 32-bit integer in filnam.
- K$UPOS — Set the current position in the UFD from the 32-bit integer in filnam.

funit — A unit on which a UFD is currently opened for reading. (A UFD may be opened with a call to SRCH$$.)

buffer — A one dimensional array into which entries of the UFD are read.

buflen — The length, in words, of buffer.

rnw — An integer variable that will be set to the number of words read.

filnam — A 32-bit integer variable used for keys of K$GPOS and K$SPOS or a name for use with K$NAME.

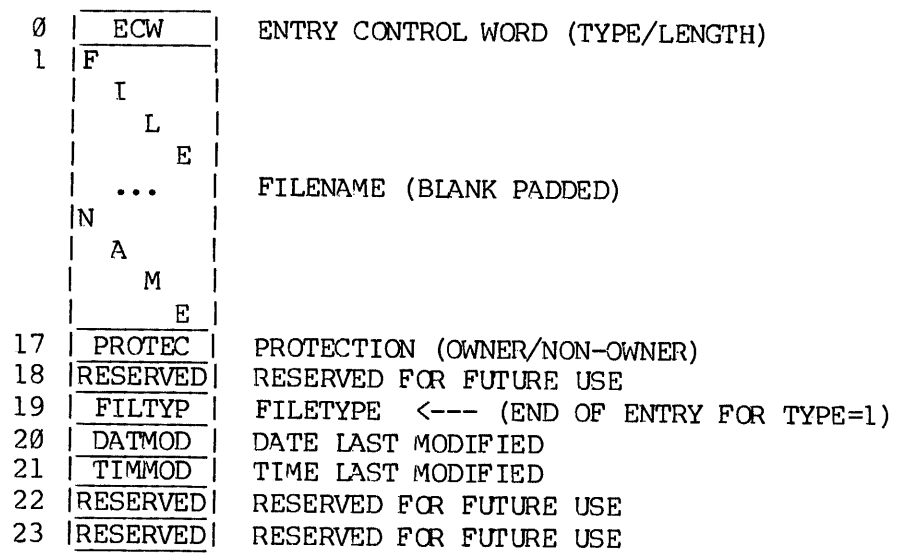
namlen — A 16-bit integer variable specifying the length in characters (Ø-32) of filnam. This variable is only used with K$NAME

code — An integer variable to be set to the return error code.

RDEN$$ is used to read entries from a UFD. rnw words are returned in buffer, and the file unit position is advanced to the start of the next entry. Return code E$EOF means no more entries, E$BFTS means buffer is too small for the entry.

In the file management system, UFDs are not compressed when files are deleted, and vacant entries may be reused. Thus, a newly created file is not necessarily found at the end of a UFD.

The complete format of currently defined entries is given here. (All numbers are decimal unless preceded by a ':'.)



ECW Entry Control Word. An ECW is the first word in any entry and consists of two 8-bit subfields. The high-order 8 bits indicate the type of the entry, the low-order 8 bits give the length of the entry in words including the ECW itself. Possible values of the ECW are as follows:

:000001 Type=0, length=1. This entry indicates either a UFD header or a vacant entry. No information other than the ECW is returned.

:000424 Type=1, length=20. Type=1 indicates an old partition UFD entry. Words 0-19 in the diagram above are returned.

:ØØ1Ø3Ø Type=2, length=24. Type=2 indicates a new partition UFD entry. All the above information is returned. Reserved fields should be ignored.

User programs should ignore any entry-types that are not recognized. This allows future expansion of the file system without unduly affecting old programs.

FILENAME Up to 32 characters of filename, blank padded.

PROTEC Owner and nonowner protection attributes. The owner rights are in the high-order 8 bits, the non-owner in the low-order 8 bits. The meanings of the bit positions are as follows (a set bit grants the indicated access right):

1-5,9-13	Reserved for future use.
6,14	Delete/truncate rights.
7,15	Write-access rights.
8,16	Read-access rights.

FILTYP On a new partition, the low-order eight bits indicate the type of the file as follows:

Ø	SAM file.
1	DAM file.
2	SAM Segment directory.
3	DAM Segment Directory.
4	UFD

On an old partition, the filetype is invalid. The file must be opened with SRCH$$ to determine its type. Of the high order 8 bits, six are currently defined as follows:

bit 1 set only for the BOOT and DSKRAT files if they are on a storage module disk.

bit 2 The dumped bit. This bit can be set by a call to SATR$$ and is reset whenever the file is modified. This bit is used by the utility program that dumps only modified files to magnetic tape. Users are normally not interested in this bit.

bit 3 This bit is set by PRIMOS II when it modifies the file and reset by PRIMOS (and PRIMOS III) when it modifies the file. If this bit is set, the time-date field for the file will not be current because PRIMOS II doesn't update the date-time stamp when it modifies a file.

bit 4 This bit is set to indicate that this is a special file. The only special files are BOOT, MFD, BADSPT, and the DSKRAT file which has the name packname. This bit, and this bit only is valid on both new and old style partitions.

bits 5-6 Setting of the per-file read/write lock.

The PRIMOS file system supports individual values of the read/write lock (RWLOCK) on a per-file basis, for those files residing on new partitions. The read/write lock is used to regulate concurrent access to the file, and was formerly alterable only on a system-wide basis.

The meaning of the lock values is:

Value	bits 5,6	Meaning
Ø	Ø,Ø	Use system-wide RWLOCK to regulate concurrent access.
1	Ø,1	Allow arbitrary readers or one writer.
2	1,Ø	Allow arbitrary readers and one writer.
3	1,1	Allow arbitrary readers and arbitrary writers.

New files are initially created with a per-file read/write lock of zero.

UFDs do not have user-alterable read/write locks, though segment directories do. Files in directory have the per-file read/write lock of the segment directory.

The per-file read/write lock value is read by RDEN$$. It is set by a SATR$$ call with a key of K$RWLK. The desired value is supplied in bits 15 and 16 of ARRAY(1), the remaining bits of which must be zero. On old partitions, the SATR$$ call fails with an error code of E$OLDP. Owner rights to the containing UFD are required, otherwise the call fails with an error code of E$NRIT. An attempt to set the lock value of a UFD fails with an error code of E$DIRE. If the SATR$$ call requests a lock value which is more restrictive than the current usage of the file, the file's lock value is changed and current users of the file are unaffected, but any new openings subsequently requested are governed by the new lock value. It is unspecified what happens when bits 1-13 of ARRAY(1) are not zero.

The commands MAGSAV and MAGRST properly save and restore the per-file read/write lock along with the file itself. Existing backup tapes without saved read/write locks on them are restored with read/write locks of zero, so the system-wide RWLOCK setting continues to control access to such files.

The FUTIL command copies the per-file read/write lock setting along with the file when performing a TRECPY of a UFDCPY (but not a COPY) operation. FUTIL prints the value of the per-file read/write lock if the option RWLOCK is specified to the LISTF request.

DATMOD The date on which the file was last modified. The date, which is valid only on new partitions, is held in the binary form YYYYYYYMMMMDDDDD, where YYYYYYY is the year modulo 100, MMMM is the month, DDDDD is the day.

TIMMOD The time at which the file was last modified. The time, which is valid only in new partitions, is held in binary seconds-since-midnight divided by four.

Examples

1. Read next entry from new or old UFD:

```
100   CALL RDEN$$ (K$READ,funit,ENTRY,24,RNW,0,0,CODE)
      IF (CODE .NE. 0) GOTO <error handler>
      TYPE=RS(ENTRY(1),8)  /* GET TYPE OF ENTRY JUST READ
      IF (TYPE.NE.1.AND.TYPE.NE.2) GOTO 100 /* UNKNOWN
```

2. Position to beginning of UFD:

```
CALL RDEN$$ (K$UPOS,funit,0,0,0,000000,0,code)
```

▶ RDLIN$

RDLIN$ reads a line of characters from a compressed or uncompressed ASCII disk file.

CALL RDLIN$ (funit, buff, count, code)

funit A file unit (1-126) on which the file to be read is open.

buff An array of count words in which the line of information from the disk file is to be read.

count The size of buff in words.

code A return variable set to 0 if no errors, or an error code if an error. See PRWF$$ for a list of possible error codes.

A line of characters from File Unit funit is read into Buffer buff, two characters per word. Lines on the disk are separated by the new line character. The character DC1 (221 octal) followed by a character count when read from the disk is replaced by character-count blanks. If the line on the disk is less than 2*count characters, the remaining space in buff is filled with blanks. If the line on the disk is greater than 2*count characters, only 2*count characters fill buff and the remaining characters on the disk file line are ignored. In all cases, the new line never appears as part of the line in buff. RDLIN$ is the same routine as I$ADO7 except that the altrtn argument has been replaced by the code argument.

▶ REST$$

REST$$ reads an R-mode memory image from a file in the current UFD into memory. The SAVE'd parameters for a file previously written to the disk by the SAVE or SAVE$$ subroutine or the SAVE command are loaded into the nine word array vector. The memory image itself is then loaded into memory using the starting and ending addresses provided by vector(1) and vector(2).

CALL REST$$(vector,filnam,namlen,code)

vector — A nine word array set by REST$$. vector(1) is set to the first location in memory to be restored. vector(2) is set to the last location to be restored. The rest of the array is set as follows:

vector(3) saved P register
vector(4) saved A register
vector(5) saved B register
vector(6) saved X register
vector(7) saved Keys
vector(8) not used
vector(9) not used

filnam — The name of the file containing the memory image.

namlen — The length in characters (1-32) of filnam.

code — An integer variable set to the return code.

Note

Use the PRIMOS command SEG to restore V-mode memory image from a file.

▶ RESU$$

RESU$$ restores an R-mode memory image from a file in the current UFD, initializes registers from the saved parameters, and starts executing the program.

CALL RESU$$(filnam,namlen)

filnam The name of the file containing the memory image.

namlen The length (1-32) in characters of filnam.

RESU$$ does not have a code argument. If an error occurs, an error message is typed and control returns to command level.

▶ SATR$$

SATR$$ allows the setting or modification of a file's attributes in its UFD entry.

CALL SATR$$ (key,filnam,namlen,array,code)

key An integer variable specifying the action to take. Possible values are:

K$PROT Set protection attributes from array(1). array(2) is ignored for old partitions and must be Ø for new partitions (it is reserved for expansion). The meaning of the protection bits in array(1) is given under the description of RDEN$$.

K$DTIM Set date/time modified from array(1) and array(2). The format of the date/time is given under the description for RDEN$$.

K$DMPB Set the dumped bit. This bit is set by the utility program that dumps modified files and is reset by the operating system whenever the file is modified. Users should not use this key.

K$RWLK Users can set the per-file read/write lock on a per-file basis. Bits 15 and 16 of array(1) are set by the user for specific lock values. Refer to RDEN$$ for further information on the read/write lock.

Note

The date-time-modified and the dumped bit are modified by PRIMOS. When these fields are changed for a file, the date-time-modified field of the UFD containing that file (parent UFD) is not changed. However, when the name or protection attributes of the file are changed, the date-time-modified and the dumped bit of the parent UFD are updated; and the dumped bit for the file is reset.

filnam	The name of the file whose attributes are to be modified. The current UFD is searched for filnam.
namlen	The length in characters of filnam.
array	A two-word array containing the attributes. For K$PROT, array(2) must be zero.
code	An integer variable set to the return code.

Owner rights are required on the UFD containing the entry to be modified.

The formats of the attributes in array are the same as those in a UFD entry obtained from RDEN$$.

An attempt to set the date/time modified, the dumped bit, or the read-write lock on an old partition will result in an E$OLDP error (error message 'OLD PARTITION').

Since a call to SATR$$ modifies the UFD, the date/time modified of the UFD itself is updated.

Examples

1. Set default protection attributes on MYFILE:

```
ARRAY(1)=:3400 /* OWNER=7, NON-OWNER=0
ARRAY(2)=0 /* SECOND WORD MUST BE 0
CALL SATR$$ (K$PROT,'MYFILE',6,ARRAY(1),CODE)
```

2. Set both owner and non-owner attributes to read-only (note carefully bit positioning in two-word octal constant):

```
CALL SATR$$ (K$PROT,'NO-YOU-DON''T',12,:100200000,CODE)
```

3. Set date/time modified from UFD entry read into ENTRY by RDEN$$:

```
CALL SATR$$ (K$DTIM,FILNAM,6,ENTRY(21),CODE)
```

▶ SAVE$$

SAVE$$ is used to save an R-mode memory image as a file in the current UFD.

CALL SAVE$$(vector,filnam,namlen,code)

vector — A nine word array the user sets up before calling SAVE$$ vector(1) is set to an integer which is the first location in memory to be saved and vector(2) is set to the last location to be saved. The rest of the array is set at the user's option and has the following meaning:

vector(3) saved P register
vector(4) Saved A register
vector(5) Saved B register
vector(6) Saved X register
vector(7) Saved Keys
vector(8) not used
vector(9) not used

filnam — The name of the file to contain the memory image.

namlen — The length in characters (1-32) of filnam.

code — An integer return code.

▶ SGDR$$

SGDR$$ positions in a segment directory, reads entries, and allows modification of a directory's size.

CALL SGDR$$ (key,funit,entrya,entryb,code)

key — An integer specifying the action to be performed. Possible values are:

K$SPOS — Move the file pointer of funit to the position given by the value of entrya. Return 1 in entryb if entrya contains a file, return Ø if entrya exists but does not contain a file, return -1 if entrya does not exist (is beyond EOF). If EOF is reached on K$SPOS, the file

pointer is left at EOF. The directory must be open for reading or both reading and writing.

K$FULL Move the file pointer of funit to the position given by the value of entrya. If the position contains a file, set entryb to the value of entrya. If the position is empty, search for the first non-empty entry following the position specified. If a non-empty entry exists, set entryb to the position of that entry. If the EOF is reached and a entry with a file has not been found, then return -1 in entryb. If EOF is reached on K$FULL, the file pointer is left at EOF.

K$FREE Act in the same manner as K$FULL, but find an entry that does not contain a file.

K$GOND Move the file pointer of funit to the end-of-file position and return in entryb the file entry number of the end of the file.

K$GPOS Return in entryb the file entry number pointed to by the file pointer of funit.

K$MSIZ Make the segment directory open on funit entrya entries long. The file pointer is moved to the end of file. The directory must be open for both reading and writing.

K$MVNT The entry pointed to by entrya is moved to the entry pointed to by entryb. The entrya entry is replaced with a null pointer. Errors are generated by K$MVNT if there is no file at entrya, if there is already a file at entryb, if either entrya or entryb are at or beyond EOF. The file pointer is left at an undefined position. The directory must be open for both reading and writing.

funit The file unit on which the segment directory is open.

entrya An unsigned 16-bit entry number in the directory, to be interpreted according to key.

entryb An unsigned 16-bit integer set or used according to key.

code An integer variable set to the return code.

When using SGDR$$, the segment directory must not be opened for write-only access.

A K$MSIZ call with entrya=Ø causes the directory to have no entries. If the value of entrya is such as to truncate the directory, all entries including and beyond the one pointed to by entrya must be null. See SRCH$$ for more segment directory information.

Note

When sequentially reading a directory (K$SPOS, entrya = entrya+1, K$SPOS, ...), entryb=-1 indicates the end of the directory, NOT the return code E$EOF. E$EOF is returned when entrya indicates a position beyond EOF, i.e., the entry following the first K$POS to return entryb=-1.

Examples

1. Read sequentially through the segment directory open on 6:

```
        CURPOS=-1
100     CURPOS=CURPOS+1
        CALL SGDR$$ (K$SPOS,6,CURPOS,RETVAL,CODE)
        IF (RETVAL) 200,300,400 /* BOTTOM, NO FILE, IS FILE
```

2. Make directory open on 2 as big as directory open on 1:

```
CALL SGDR$$ (K$GOND,1,Ø,SIZE,CODE)
IF (CODE.NE.Ø) GOTO <error handler>
CALL SGDR$$ (K$MSIZ,2,SIZE,Ø,CODE)
```

▶ SPAS$$

SPAS$$ sets the passwords of the current UFD.

CALL SPAS$$(opass,npass,code)

- opass — A three word array that contains the password to set as the owner password.
- npass — A three word array that contains the password to set as the nonwoner password.
- code — An integer variable set to the return code.

SPAS$$ requires owner rights to the current UFD. For passwords intended to be typed from the terminal, passwords should not start with a number nor should they contain blanks commas = ! @ { } [ ] ( ) ^ < or >. Passwords should not contain lower-case characters but may contain any other characters including control characters.

Passwords which are not intended to be typed from the terminal (i.e., access through program only) can be any bit pattern.

▶ SRCH$$

SRCH$$ is used to open a file, close a file, delete a file, or check on the existence of a file.

CALL SRCH$$ (action+ref+newfil,filnam,namlen,funit,type,code)

action — A subkey indicating the action to be performed. Possible values are:

- K$READ — Open filnam for reading on funit.
- K$WRIT — Open filnam for writing on funit.
- K$RDWR — Open filnam for reading and writing on funit.
- K$CLOS — Close file by filnam or by funit.
- K$DELE — Delete file filnam.
- K$EXST — Check on existence of filnam.

ref — A subkey modifying the action subkey as follows:

- K$IUFD — Search for file filnam in the current UFD (this is the default).
- K$ISEG — Perform the action specified by action on the file that is a segment directory entry in the directory open on file unit filnam.
- K$CACC — Change the access mode of the file already open on funit to action. (K$READ, K$WRIT, K$RDWR only).
- K$GETU — Open filnam on an unused file-unit selected by PRIMOS. The unit number is returned in funit. When this key is used, SRCH$$ supplies a unit number not currently in use. See example of use of this key later in this section.

newfil A subkey indicating the type of file to create if filnam does not exist. Possible values are:

K$NSAM New threaded (SAM) file (this is the default).

K$NDAM New directed (DAM) file.

K$NSGS New threaded (SAM) segment directory.

K$NSGD New directed (DAM) segment directory.

It is not possible to generate a new UFD with SRCH$$; use CREA$$ instead.

filnam Name of the file to be opened (2 characters per word). K$CURR can be used to open the current UFD (ACTION keys K$READ, K$WRIT, or K$RDWR only). If ref is K$ISEG, filnam is a file unit from 1 to 62 (1 to 15 under PRIMOS II) on which a segment directory is already open.

namlen The length in characters (1-32) of filnam.

funit The number (1-15 under PRIMOS II, 1-162 under PRIMOS) of the file unit to be opened or closed, or returned argument with K$GETU key.

type An integer variable that is set to the type of the file opened. type is set only on calls that open a file -- it is unmodified for other calls. Possible values of type are:

Ø SAM file
1 DAM File
2 SAM Segment Directory
3 DAM Segment Directory
4 UFD

code An integer variable set to the return code.

SRCH$$ is a complex subroutine that has multiple uses. The most common use is to open and close files.

Opening and Closing Files

Opening a file consists of connecting a file to the file unit. After a file is opened, the file may be accessed to transfer information to or from the file or to position the current position pointer of a file unit (file pointer). These actions are accomplished by other subroutines, which reference the file through the attached file unit, such as PRWF$$, SGDR$$, RDEN$$, RDLIN$, WTLIN$, I$ADØ7, O$ADØ7, RDASC,

and WRASC. Information is also transferred through the statements in specific languages, such as the READ and WRITE statements in FORTRAN.

On opening a file, SRCH$$ specifies:

1. Allowable operations that may be performed by PRWF$$ and other routines (these operations are read-only, write-only or both read and write).

2. Where to look for the file, or where to add the file if the file does not currently exist. SRCH$$ either specifies a filename in the currently attached user file directory or a file unit number on which a segment directory is open. In the segment directory reference, the file to be opened has its beginning disk address given by the entry at the current position pointer of the file unit.

Each file in a UFD has associated with it two sets of access rights, one for the owner and one for the nonowner of the UFD. These access rights are initially owner-has-all, nonowner-has none. They can be changed using the PROTECT comand or the SATR$$ subroutine. These access rights (reading allowed, writing allowed, or delete allowed, etc.) are checked on any attempt to open a file. A NO RIGHT error code (E$NRIT) is set if the user does not have the required rights.

If the file cannot be found on open for reading, SRCH$$ generates the file-not-found error code (E$FNTF). If the file unit is already in use, SRCH$$ generates the unit in use error code (E$UIUS).

The Read/Write Interlock

Under default conditions, the system allows any number of readers if there are no writers or a single writer and no readers for the same file. The system prevents one user from opening a file for writing when another user has the file open for reading or writing. The system prevents one user from opening the file for reading or writing while another user has the file open for writing. Furthermore, these interlocks hold for a single user attempting to open a file on multiple file units he has available. If the interlock is violated, the FILE IN USE error code is generated (E$FIUS).

This interlock may be changed on a per-file basis. (Refer to RDEN$$.)

On closing a file, it is possible to close by name or by file unit. SRCH$$ attempts to close by filnam unless filnam is specified as Ø in which case it closes the file unit specified. If filnam is not found, an error is generated (code = E$FNTF), but if the file unit is specified, SRCH$$ ensures that the file unit specified by funit is closed and never generates an error code (unless funit is out of range). If the file has been modified while it was open, the date-time stamp of the file is updated when the file is closed.

Changing the Access Mode of an Open File

A user may change the access mode of a file that is open on funit to open-for-reading, open-for-writing, or open for both reading and writing, using the K$CACC key. Note that access rights and the read/write interlock rules from the file are checked and the attempt to change access may fail.

Adding and Deleting Files in UFDs

A call to SRCH$$ to open a file for writing or both reading and writing, causes SRCH$$ to look in the current UFD for the file. If the file is not found in the UFD, a new file is created of Ø length and an entry for the file is put in the UFD. The date/time of the file is set to the current date/time, the access rights are set to owner-has-all-rights, nonowner-has-none, the read/write interlock is set to the system standard read/write lock and the file type to that file type specified in the SRCH$$ call. If the file type is not specified, it is a SAM file. Note that nonowners cannot generate new files (error code returned is E$NRIT).

A call to delete a file must specify a legal funit although the file system does not use that file unit during the delete. Deleting a file returns the records of the file to the DSKRAT pool of free records and erases the entry from the UFD leaving a vacant hole. Vacant holes in UFDs will be reused for new files if of the right size, so new files do not always appear at the end of your UFD. These vacant holes take very little room on the disk in most cases. These holes are compressed out of UFDs when the FIXRAT maintenance program is run by the system operator. See The System Administrator's Guide (PDR31Ø9).

Checking the Existence of a File

If the user wishes to find out if a certain file exists in the current ufd or segment directory, the SRCH$$ K$EXST key can be used. The file is not affected in any way and access rights and the read/write interlock are not checked.

Operations on Files that are UFDs

Files in the current UFD that are subUFDs can be opened only for reading. The contents of entries of subUFDs can be read through calls to RDEN$$ and GPAS$$ once the subUFD is open. The current UFD can be opened for reading by specifying the key K$CURR in the filnam field of the SRCH$$ call. Calls to the SATR$$ or SPAS$$ subroutines require that the current UFD not be open or the FILE IN USE error is generated. New UFDs can only be created using the CREA$$ subroutine, not SRCH$$. UFDs may be deleted with SRCH$$ only if the UFD contains no files. The FUTIL command can delete a nested structure of UFDs.

Operations Involving Segment Directories

Segment directories are directories in which the files are referenced by their position in the directory rather than a name. Furthermore, the directory entry associated with a file contains the attributes such as date/time, protection or the read/write lock, of the highest level segment directory in the UFD. Segment directories are not attached but are operated on using SRCH$$ and SGDR$$.

To create a segment directory, use SRCH$$ to open a new file for reading and writing with the file type specified as SAM segment directory or DAM segment directory.

With the file open, use SGDR$$ to make the segment directory contain a certain number of null file entries (K$MSIZ key).

To create a file in a segment directory, first open the directory for reading and writing on a funit (e.g. SUNIT) if it is not already open. Next, use SGDR$$ to position to the null file entry desired. Next, use SRCH$$ to open a new file for writing or reading and writing in the segment directory by using the K$ISEG reference key and placing the SUNIT number of the segment directory in the filnam argument. The file unit of the new file goes in the usual field (funit). SRCH$$ will create the new file and place a pointer to the new file in the segment directory entry of SUNIT.

Use SRCH$$ to close by unit or name (with K$ISEG) a file in a segment directory.

To open a file that already exists in a segment directory, open the segment directory and position to the desired entry as explained above. Use SRCH$$ to open the file as explained above. If the directory entry already contains a pointer to the file, that file will be opened. If not, and the attempt is to open for reading, the file not found error is generated. Any type of file except a UFD may be created in a segment directory.

To delete a file in a segment directory, open the segment directory, position to the file desired, then use SRCH$$ with the K$ISEG and K$DELE keys. SRCH$$ returns the record of the file to the DSKRAT and replaces the pointer to the file with a null pointer in the segment directory entry.

Finally, to delete a segment directory, the user must first delete all files in the directory, set the size of the directory to Ø using SGDR$$, close the directory, then delete it with SRCH$$. The FUTIL command may be used to delete a segment directory at command level.

Files in a segment directory have the protection attributes of the directory. The date/time field of the directory reflects the latest change made to the directory or any file in the directory.

Filenames

Filenames may be 1-32 characters in length, the first character of which must not be numeric. Filenames can be composed only of the following characters: A-Z Ø-9 # _ $ & * - . and /. Filenames may not contain embedded blanks; filenames may be specified with trailing blanks. An attempt to create a file with an invalid filename results in the error code E$BNAM (illegal name).

Examples

1. Open new SAM file named RESULTS for output on file unit 2:

```
CALL SRCH$$(K$WRIT,'RESULTS',7,2,TYPE,CODE)
```

2. Create new DAM file in the segment directory open on SGUNIT and open for reading and writing on DMUNIT:

```
CALL SRCH$$(K$RDWR+K$ISEG+K$NDAM,SGUNIT,1,DMUNIT,TYPE,CODE)
```

3. Close and delete the file created in the above call:

```
CALL SRCH$$(K$CLOS,Ø,Ø,DMUNIT,Ø,CODE)
CALL SRCH$$ (K$DELE+K$ISEG,SGUNIT,Ø,Ø,Ø,CODE)
```

4. See if filename 'MY.BLACK.HEN' is in current UFD:

```
CALL SRCH$$ (K$EXST+K$IUFD,'MY.BLACK.HEN',12,Ø,TYPE,CODE)
IF (CODE.EQ.E$FNTF) CALL TNOU('NOT FOUND',9)
```

5. Create a new segment directory and a new SAM file as its first entry:

```
CALL SRCH$$(K$RDWR+K$NSGS,'SEGDIR',6,UNIT,TYPE,CODE)
CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,UNIT,Ø,7,TYPE,CODE)
```

6. Open the file named 'FILE' in the user's currently attached UFD:

```
 CALL SRCH$$(K$READ+K$GETU,'FILE',4,UNIT,TYPE,
X            CODE)
 IF (CODE .NE. Ø) GOTO error_processor
```

The above FORTRAN call will attempt to open the file named 'FILE' in the user's currently attached UFD. If successful, the file unit number on which 'FILE' has been opened is returned in UNIT. The type of the file opened is returned in TYPE, and CODE is set to zero if there are no errors. If there are any errors, CODE will be nonzero, and the values of TYPE and UNIT are undefined.

If no file units are available, the error code E$FUIU (all units in use) is returned. This code is returned if either the process (user) has exceeded the maximum number of file units the process (user) may have, or the total number of file units in use for all processes (users) exceeds the maximum number of file units available to all processes (users).

▶ TSRC$$

TSRC$$ AND TREENAMES

TSRC$$ is a subroutine to open a file anywhere in the PRIMOS file structure.

CALL TSRC$$ (action+newfil, treename, funit, chrpos, type, code)

action — A subkey indicating the action to be performed. Possible values are:

- K$READ Open treename for reading on funit.
- K$WRIT Open treename for writing on funit.
- K$RDWR Open treename for reading and writing on funit.
- K$DELE Delete file treename.
- K$EXST Check on existence of treename.
- K$CLOS Close treename (not funit).

newfil — A subkey indicating the type of file to create if treename does not exist. Possible values are:

- K$NSAM New threaded (SAM) file (this is default).
- K$NDAM New directed (DAM) file.
- K$NSGS New threaded (SAM) segment directory.
- K$NSGD New directed (DAM) segment directory.

treename — A specification of any file in any directory or subdirectory stored in array treename packed two characters per word.

funit — The number (1-126) of the file unit to be opened or deleted. funit is closed before any action is attempted.

chrpos A two element integer array setup as follows:

chrpos(1) On entry, set to contain the first character in the array that is part of the treename, the count starting at Ø. On exit, it will be pointing one past the last character that was part of the treename. A comma, new line, or carriage return will terminate the name, as will end of array. In case of error, chrpos(1) points one past the treename component that caused the error. chrpos(1) is always modified by this subroutine, therefore, must be set up before each call.

chrpos(2) The number of characters in the treename array.

type An integer variable set to the type of the file opened. type is set only on calls that open a file; it is unmodified for other calls. Possible values for type are:

Ø	SAM File
1	DAM File
2	SAM Segment Directory
3	DAM Segment Directory
4	UFD

code An integer variable set to the return code. If no errors, code is Ø.

TSRC$$ always closes the specified file unit then attaches to the user's home UFD before attempting any action. If the user's home UFD differs from his current UFD before calling TSRC$$, he will find himself attached to his home UFD following the call. See SRCH$$ for more details on file manipulation.

Caution

Do not use TSRC$$ to perform a change access (K$CACC).

Treename Definition

A treename is a syntax convention that allows the specification of any file in any directory or subdirectory. A treename may be used to open or delete a file using subroutine TSRC$$. Treenames may also be used in place of simple filenames in most external commands such as SLIST. Treename as used here, is synonymous with "pathname" as described in the PRIMOS Commands Guide.

The simplest form of a treename is a simple file name as allowed by SRCH$$. The file is assumed to be located in the home directory.

The general form is a starting directory specifier, zero, one, or more subdirectory specifiers, and then the file name.

The starting directory specifier has the following formats (square brackets ([]) indicate an optional item):

1. UFDname [password]>

2. *>

3. <volumename> UFDname [password]>

4. <logical-disk-number> UFDname [password]>

In form 1, all MFDs are searched for the named directory in logical disk order.

In form 2, the home directory is the starting directory.

In form 3, the volume with the specified name is searched for the specified UFD name. If the volume name is a single asterisk (*), the MFD in the home volume is searched.

In form 4, the volume with the specified octal logical disk number is searched for the specified UFD name.

A subdirectory specifier has the following format:

ufdname [password]>

The UFD is assumed to be in the directory specified by the preceding specifier. Spaces are not significant except that they may not occur within a name and must separate a UFD from its password. If a name is longer than 32 characters, the excess characters are ignored.

Examples

ABC	File named ABC in home directory.
XYZ>ABC	File named ABC in UFD=XYZ.
<INV>XYZ>ABC	File named ABC in UFD=XYZ on volume =INV.
<*>XYZ>ABC	File named ABC in UFD=XYZ on home volume.
<5>XYZ>ABC	File named ABC in UFD=XYZ on logical disk 5.
*>XYZ>ABC	File named ABC in subUFD=XYZ in home directory.

*>XYZ>IJK>ABC File named ABC in subUFD IJK in subUFD=XYZ in home directory.

XYZ DEF>ABC File named ABC in UFD=XYZ with password =DEF.

Treenames specified as parameters to external commands should not contain spaces, as the space or comma is used to separate one parameter from another. If a space must be specified due to a password, enclose the entire treename in single quotes.

▶ UPDATE

CALL UPDATE (key,Ø)

key 1

Update current UFD; DSKRAT buffers to disk, if necessary; and undefine DSKRAT in memory.

This call is not normally used. This call is effective only under PRIMOS II. Under PRIMOS III or PRIMOS it has no effect.

▶ WTLIN$

WTLIN$ writes a line of characters in ASCII format to a file in compressed ASCII format.

CALL WTLIN$(funit, buffer, count, code)

funit A file unit (1-126) on which the file to be written is open for writing.

buffer An array of count words from which the line of characters is to be written. It should contain 2 characters per word

count The size of buffer in 16-bit words.

code A return variable set to Ø if no errors, or an error code if an error has occurred. Refer to Appendix G for a list of error codes.

Information is written on the disk in compressed ASCII format. Multiple blank characters are replaced by the character DC1 (271 octal) followed by a character count. Trailing blanks are removed and the end of record is indicated by adding a new line character, or a new line character followed by null. WTLIN$ is the same routine as O$AD07, except the altrtn argument has been replaced by the code argument.

SECTION 5

MISCELLANEOUS PRIMOS SUBROUTINES

This section describes subroutines which perform miscellaneous PRIMOS functions. The PRIMOS routines described in this section are: BATCH$, BREAK$, C1IN, CL$GET, CNIN$, COMANL, DUPLX$, ERKL$$, ERRPR$, EXIT, GINFO, LOGO$$, PHANT$, RDTK$$, RECYCL, TEXTO$, TIMDAT.

▶ BATCH$

BATCH$ starts a phantom user. BATCH$ is the same as PHANT$, but has the additional function of starting a phantom user under a different login name (usrname). BATCH$ is called by a procedure running under control of the supervisor (user 1) or a phantom initiated by user 1.

CALL BATCH$ (fname, fnlen, unit, usrnam, unlen, user, code)

fname Array containing name of command input file to be started as a phantom (INTEGER*2).

fnlen Length (in characters) of fname (INTEGER*2).

unit File unit on which to open fnam. If value specified is Ø, default is unit 6 (INTEGER*2).

usrnam User name the phantom is to be started under. (INTEGER*2)

unlen Length of usrnam (INTEGER*2).

user User number that usrnam was assigned.

code Code returned to the user, indicating any errors (INTEGER*2). Possible values are:

Ø No error.

E$NRIT Not called from process initiated from system console or insufficient access rights to fname.

E$DIRE Fname is directory, not a file name.

E$NPHA No phantoms available.

▶ BREAK$

BREAK$ inhibits or enables CONTROL-P for interrupting a program.

CALL BREAK$ (.TRUE.)
CALL BREAK$ (.FALSE.)

The LOGIN command initializes the user terminal so that the CONTROL-P or BREAK key cause an interrupt. Under PRIMOS III and PRIMOS, the BREAK$ routine, called with the argument .FALSE., enables the CONTROL-P or BREAK key to interrupt a running program.

On the other hand, the BREAK$ routine called with the argument .TRUE., inhibits the CONTROL-P or BREAK characters from interrupting a running program.

This routine maintains a per-user QUIT inhibit master list. Each call to BREAK$ to inhibit or enable QUIT increments or decrements a counter. QUITs are enabled only when the counter is zero; i.e., the counter goes positive with inhibit and cannot be decremented below zero.

Under PRIMOS II, BREAK$ has no effect.

▶ ClIN

This routine gets the next character from the terminal or a command file, depending upon the source of the command stream.

CALL ClIN (char)

The next character is read or loaded into char (right-justified and zero-filled). If the character is .CR., char is set to .NL. (new line).

Line feeds are discarded by the operating system, and are not detected by the ClIN subroutine.

▶ CL$GET

CL$GET reads a single line of input text from the currently defined command input stream. The line is returned as a varying character string without the newline character at the end. An empty command line or one consisting of all blanks will compare equal to the null string.

CALL CL$GET (comline, comlinesize, status)

comline	Varying character string into which the text will be read from the command input stream.
comlinesize	Maximum length, in characters, of comline. Because comline is a varying string, it is not blank padded to this size.
status	Return error code.

▶ CNIN$

This subroutine is the raw data mover used to move a specified number of characters from the terminal or command file to the user program's address space.

CALL CNIN$ (buffer, char-count, actual-count)

buffer	A buffer in which the string of characters read from the input stream are to be placed (two characters per word).
char-count	The number of characters to be transferred from the input stream to the buffer specified by buffer.
actual-count	A return argument. It specifies the number of characters read by the call to CNIN$. If reading continues until a new line character is encountered, the count includes the line character.

CNIN$ reads from the input stream until either a NEW LINE character is encountered or the number of characters specified by the char-count argument are read. Characters are left-justified, and if an odd number of characters are read, the remaining character space is not zero or blank filled. The question mark and quotation mark characters are not interpreted.

Input to CNIN$ is obtained from the terminal unless the user has previously given the COMINPUT or PHANTOM commands, and these commands are still in control. The COMINPUT or PHANTOM commands switch the input stream so that it comes from a file rather than the terminal (refer to the PRIMOS Commands Reference Guide (FDR3108) for further information).

▶ COMANL

COMANL causes a line of text to be read from the terminal or from a command file, depending upon the source of the command stream.

CALL COMANL

The line is read into a supervisor text buffer. This buffer may be accessed by a call to RDTK$$. The supervisor text buffer holds 80 characters. The supervisor text buffer is also used by CNIN$ and T$AMLC. The contents of this buffer must be picked up by RDTK$$ after a call to COMANL and before calls to CNIN$ or T$AMLC.

▶ DUPLX$

The DUPLX$ subroutine is called to control the manner in which the operating system treats the user terminal.

CALL DUPLX$ (mode)

It returns the terminal configuration word and internal buffer number as the value of the function. In addition, if the mode passed to DUPLX$ is equal to -1, no updating of the configuration word takes place. In this case, the current value is returned. DUPLX$ must be declared as an INTEGER function if the returned value is to be used by the calling program. Values for mode are:

Bit	Mask	Meaning if Bit is SET
1	100000	Half duplex
2	040000	Do not echo LINE-FEED after CARRIAGE RETURN.
3	020000	Turn on X-OFF/X-ON character recognition.
4	010000	Output currently suppress (X-OFF received).
5-8	007400	Reserved.
9-16	000377	Internal buffer number (read-only).

DUPLX$ has no effect under PRIMOS II.

The mode of a user terminal is not affected by the LOGIN or LOGOUT commands.

The mode of the user terminal may also be set at the supervisor terminal by using the AMLC command.

User may use the PRIMOS TERM command to change their terminal characteristics.

▶ ERKL$$

The ERKL$$ subroutine reads or sets erase and kill characters.

CALL ERKL$$(key,erase,kill,code)

key — A parameter specifying the action to be taken. Possible values are:

- K$WRIT — Set erase and kill characters.
- K$READ — Read erase and kill characters.

erase — On key K$WRIT, the character contained in the right byte of erase replaces the operating system's per-user erase character. If erase is Ø, no action takes place. On key K$READ, the current per-user system erase character is placed in erase, right-justified with leading zeros.

kill — On key = K$WRIT, the character contained in the right byte of kill replaces the operating system's per-user kill. The current per-user system kill character is placed in kill right justified with leading zeros.

code — An integer variable set to the return code. Possible values are:

- Ø — If no errors.
- E$BPAR — If attempt to set characters is improper.

Erase and kill characters are reset to default values upon a logout or login.

Erase and kill characters are interpreted by commands to the operating system and through the subroutines COMANL, RTK$$, RDCOM, RDASC, I$AA12, and I$AAØ1. All language processors and I/O statements call RDASC to get terminal input and, therefore, are affected.

RDCOM, RDASC, I$AA12, and I$AAØ1 are library subroutines that read the system's per-user erase and kill character only once when they are first invoked. Therefore, changing the erase and kill characters after a call to those subroutines does not affect erase and kill processing in these subroutines until the next program is invoked. The main purpose for users calling the ERKL$$ subroutine is to read or set these characters when the user programs do their own erase and kill processing.

Under PRIMOS II, the erase and kill characters may be read but any attempt to set them is ignored. The erase and kill characters may be set at command level by the PRIMOS TERM command.

▶ ERRPR$

ERRPR$ interprets a return code and, if non-zero, prints a standard message associated with the error return code, code, followed by optional user text. See Appendix G for more details on error handling.

CALL ERRPR$ (key,code,text,txtlen,filnam,namlen)

key — An integer specifying the action to take subsequent to printing the message. Possible values are:

- K$NRTN — Exit to the system, never return to the calling program.
- K$SRTN — Exit to the system, return to the calling program following an 'S' command.
- K$IRTN — Return immediately to the calling program.

code — An integer variable containing the return code from the routine that generated the error. If code is Ø, ERRPR$ always returns immediately to the calling program and prints nothing.

text — A message to be printed following the standard error message. Text is omitted by specifying both text and txtlen as Ø.

txtlen — The length in characters of text.

filnam The name of the program or subsystem detecting or reporting the error. filnam is omitted by specifying both filnam and namlen as Ø.

namlen The length in characters of filnam.

▶ EXIT

The EXIT subroutine provides a way to return from a user program to PRIMOS; it prints OK, (or OK:) at the terminal and PRIMOS awaits a user command.

CALL EXIT

The user may open or close files or switch directories, and restart a FORTRAN program at the next statement by typing S (i.e., START).

▶ GINFO

GINFO indicates whether or not the user is running under PRIMOS II. If running under PRIMOS II, GINFO shows where PRIMOS II is loaded in the user address space.

CALL GINFO (xervec, n)

GINFO moves n words from the supervisor into a buffer specified by xervec.

Information for PRIMOS II:

xervec word	Content
1	Low boundary of PRIMOS II buffers (77777 octal if 64K PRIMOS II).
2	High boundary of PRIMOS II (77777 octal if 64K PRIMOS II).
3	(not valid)
4	(not valid)
5	Low boundary of PRIMOS II and buffer (64K PRIMOS II only).
6	High boundary of 64K PRIMOS II.

Information for PRIMOS III and PRIMOS:

xervec word	Content
1	Ø
2	Ø
3-6	(not valid)

▶ LOGO$$

LOGO$$ logs out a user. The routine can be used by the supervisor terminal (user 1) to log out any user, or a user program may log out any process it may have started.

CALL LOGO$$ (key, user, usrnam, unlen, reserv, code)

key Operation to be performed (INTEGER*2). Possible values are:

- -1 log out all users (supervisor only).
- Ø log out (same as LOG OUT command).
- 1 log out (same as LOG OUT - NN).
- 2 log out specific user by name (supervisor or his phantoms only).

user User number to be logged out. This value is examined only if key > Ø. (INTEGER*2).

usrnam Name of user to be logged out; must correspond to number supplied in user. This value is examined only if key = 2. (INTEGER*2).

unlen Length of usrnam (in characters). This value examined only if key = 2. (INTEGER*2).

reserv Reserved for future use (INTEGER*4).

code Error code returned to user (INTEGER*2). Possible values are:

Ø	No error
E$BKEY	Bad key
E$BPAR	Invalid number specified in user.
E$BNAM	Username does not correspond to user.
E$NRIT	Attempt to log out user with name different from requestor.

▶ PHANT$

PHANT$ starts a phantom user.

CALL PHANT$ (filnam,namlen,unit,user,code)

filnam Name of command input file to be run by the phantom.

namlen Length of characters of filnam.

unit File unit on which to open filnam. If unit is Ø, unit 6 will be used.

user A variable returned as the user number of the phantom.

code The return code. If Ø, the phantom was initiated successfully. If code = E$NPHA, no phantoms were available. Other values of 'code' are file system error indications.

▶ RDTK$$

The subroutine RDTK$$ parses the command line most recently read by a call to COMANL. If no previous calls to COMANL have taken place, RDTK$$ parses the last command line typed at PRIMOS command level by the user. Parsing proceeds on a token by token basis. A command line consists of tokens (or words) separated by delimiters. The current delimiters are space, comma, /*, and newline. The characters ()`[]!{};^"?:~|\.DEL. are reserved in command lines for future use. However, one of these characters may be included in a token by enclosing the token in single quotes; for example, 'naughty(token)'. The characters /*, if unquoted, begin a comment field that extends to the end of the line and are ignored by RDTK$$.

Each call to RDTK$$ reads a single token from the command line. RDTK$$ returns the literal text of the token, together with some additional information about it. If the token is numeric, RDTK$$ will provide results of decimal and octal conversion attempts. RDTK$$ will also inform the caller if a numeric token can be interpreted as a register setting (octal parameter) under the old PRIMOS command line structure.

Do not make calls to T$AMLC or CNIN$ or calls to subroutines that call these such as FORTRAN formatted READ statement to the terminal, before parsing the command line since these subroutines cause the replacement of the information in the per-user supervisor buffer holding the command line.

CALL RDTK$$(key,info,buffer,buflen,code)

key The action to be taken by RDTK$$. Possible values are:

1 Read next token, convert to uppercase.
2 Read next token, leave in lowercase.
3 Reset to start of command line.
4 Read remainder of command line as raw text.
5 Initializes the command line.

info Set to contain the following information: (Only info(2) is set for a key value 4.)

info(1): the type of the token. Possible values are:

1 Normal token (results of numeric conversions returned).

2 (non-ignored) register setting parameter.

5 Null token.

6 End of line.

info(2): The length in characters of the token. A null token has a Ø length.

info(3): Further information about the token. The following bits of info(3) have the indicated meaning when set:

bit 1 (:1ØØØØØ) - decimal conversion successful (with no overflow), value returned in info(4).

bit 2 (:040000) - octal conversion successful, value returned in info(5). This bit always set when token type is 2.

bit 3 (:020000) - token begins with unquoted minus sign, i.e., token may be a keyword argument.

bit 4 (:010000) - this flag means that an explicit position for a register setting was given; position value returned in info(4).

bits 5-16: reserved for future use.

info(4): Contents depends on flags set in info(3). If bit 4 is set, info(4) is the position number for the register setting. (Note that if token type is 2 and bit 4 is not set, the position is implicit and must have been remembered by the caller). If bit 1 is set, info(4) is the converted decimal value. Else info(4) is undefined.

info(5): Contents depends on flags in info(3). If bit 2 is set, info(5) is the converted octal value. Else info(5) is undefined.

info(6)-info(8): reserved for future use.

buffer An array into which the literal text of the token is written by RDTK$$, two characters per word and blank-padded to length buflen (words).

buflen Is the specified length, in words, of buffer. buflen must be >= 0.

code A standard error code returned. Possible values are:

0 No errors.

E$BKEY Value of key is illegal.

E$BPAR Bad parameter; buflen is less than 0.

E$BFTS Buffer too small to contain the full text of the token. The token is truncated.

Delimiters

Delimiter characters have four functions: token separation, content indication, literal text delineation, and line termination. The set of delimiter characters is:

SP , ' NL /*

The meanings of these characters is as follows:

Blank Interpretation (SP): A single blank terminates a token. A multi-blank field is precisely equivalent to a single blank. Blanks surrounding another delimiter are ignored. Leading and trailing blanks on the command line are ignored.

Comma Interpretation: A single comma terminates a token and is equivalent to a blank. Two or more commas in succession, however, will generate null tokens. If a comma is the first or last character on the command line, a null token will be generated. A command line consisting of just n commas (with no text) will generate n+1 null tokens.

Literal Text Character ('): Literal text strings start and end with single apostrophes. Any characters, including delimiters but excluding a newline can appear inside a literal string; the entire string is treated as a single token. Rules for literal apostrophes are the same as FORTRAN's: each literal apostrophe in the string must be doubled:

'HERE''S A LITERAL ''.'

A token can be partially literal, for example, ABC'DEF'. Numbers in literal text are interpreted as textual characters (see token definitions below). A literal string is ended either with a single apostrophe or by a newline.

Newline Delimiter (NL): A newline character terminates the preceding token. If the newline is in a literal text field, the literal is terminated. If a newline is encountered before any token text or delimiter, an End-of-Line token is generated.

Comment Delimiter (/*): When the character pair /* is encountered, all subsequent text on the command line is ignored. A /* in the beginning of a command line will cause an immediate End-of-Line token to be generated.

Tokens

A token is any string of characters not containing a delimiter. A token can be from Ø to 8Ø characters in length. The following are examples of valid tokens:

```
FTN
LONG-FILENAME
1/7Ø7
6
98
String.even.Longer.than.thirty-two.characters
<tree>name
.NULL. (null string)
```

Literal text including delimiters can be entered in apostrophes using FORTRAN rules:

```
'STRING WITH EMBEDDED BLANKS'
'HERE''S A LITERAL APOSTROPHE'
```

Token Types

Associated with each token is a type. Possible token types are discussed in the following paragraphs.

Normal Token: A normal token is any string of characters except a register setting token. The string may or may not include literal text. Examples of normal tokens are:

```
FTN
AOOOl
This.is.a.token.
PARTIALLY' L I T E R A L'
'8'xxx  (Note: '8' is treated as a non-numeric.)
''''''''     (= '''')
```

Register Setting Token: Register-setting tokens (octal parameters) are now considered obsolete. They are handled by RDTK$$ solely to permit existing software and command files to continue to function. New software should not use such parameters; symbolic keywords should be used instead, as in FTN XX -64V instead of FTN XX 2/4ØØ.

The rules for recognition of a register setting parameter as such are as follows. A token of the form octal/octal is always recognized as a register setting (unless enclosed in quotes). Initially, unembellished octal integers are also recognized as implicit-position register settings. If a token beginning with an unquoted minus sign, and which does not successfully convert as a decimal integer, is found, recognition of implicit-position register settings is disabled. Recognition is re-enabled only by a subsequent occurrence of an explicit-position register setting: octal/octal.

Null Token: A null token is generated when two delimiters are encountered in a row (except for multiple context characters). Command lines generating null tokens are the following:

```
,              (Start of line is a delimiter in this case.)
X,,Y
```

End of Line Token: This token is generated when the end of the command line is reached.

Usage

RDTK$$ maintains an internal pointer that points to the next character in the command line to be scanned. This pointer is set to the start of the command line by COMANL. It can also be reset to the start of the line with a RESET (key=3) call to RDTK$$.

Following a PRIMOS command, the internal pointer is positioned after the main command. If RESUME were the command, it is positioned after the RESUME filename.

Regardless of the token type, RDTK$$ always returns the literal text of the token. Delimiter characters (unless inside apostrophes) are never returned.

If a token is truncated (too long to fit in buffer), the next call to RDTK$$ will return the next token, not the truncated text.

For register setting tokens (octal parameters), the octal position number is returned by RDTK$$ only if explicitly given in the token (e.g. 6/123). Hence, the current register setting position must be remembered by the caller.

A buflen of Ø can be used to skip over a token. The error code E$BFTS will be returned.

For key=4 (Read Raw Text), all text between the current RDTK$$ pointer and the end of the command line (newline) is returned. No checking is done for any delimiters or special characters other than newline. No forcing to upper case is performed.

▶ RECYCL

The RECYCL subroutine is called under PRIMOS to tell the system to cycle to the next user. It is a "I have nothing to do for now" call. Under PRIMOS II, RECYCL does nothing.

CALL RECYCL

Caution

Do not use this subroutine to simulate a time delay.

▶ TEXTO$

TEXTO$ checks a filename for valid format.

CALL TEXTO$ (filnam,namlen,trulen,textok)

filnam	An array containing the filename to be checked.
namlen	The length of filnam in characters.
trulen	An integer set to the true number of characters in filnam. trulen is valid only if textok is .TRUE.
textok	A logical variable set to .TRUE. if filnam is a valid filename, otherwise set to .FALSE.

trulen is the number of characters in filnam preceding the first blank. If there are no blanks, trulen is equal to namlen. See SRCH$$ for filename construction rules.

For example, to read name from terminal, check for validity, and set trulen to actual name length:

```
CALL I$AA12 (Ø,BUFFER,8Ø,$999)
CALL TEXTO$ (BUFFER,32,TRULEN,OK) /* SET TRULEN
IF (.NOT.OK) GOTO <bad-name>
```

▶ TIMDAT

TIMDAT returns the date, time, CPU time, and disk I/O time used since LOGIN, the users unique number on the system, and his login UFD name in an array as follows:

CALL TIMDAT (array, num)

array (1) Two ASCII characters representing month. Example: 11
(2) Two ASCII characters representing day. Example: 30
(3) Two ASCII characters representing year
Example: 75
(4) Integer time in minutes since midnight.
(5) Integer time in seconds.
(6) Integer time in ticks.
(7) Integer CPU time used in seconds.
(8) Integer CPU time used in ticks
(standard is 330 ticks/second).
(9) Integer disk I/O time used in seconds. (see Note)
(10) Integer disk I/O time used in ticks. (see Note)
(11) Integer number of ticks per second.
(12) User number.
(13) Six-character login name, left-justified.
(14)
(15) Example: MSMITH

num Words of array are set. This routine does not return any useful information under PRIMOS II.

Disk I/O time is from start of seek to end of transfer, including both explicit file I/O and paging operations. CPU time used in controlling the transfer is counted under CPU time, array(7) and array(8).

SECTION 6

SAMPLE PROGRAMS

This section contains sample programs illustrating the use of the file system subroutines. The programs are:

- Writing a SAM file
- Writing a DAM file
- Reading a SAM or DAM file
- Creating a segment directory
- Reading a logical record from a file
- Reading a file in a segment directory

▶ WRITING A SAM FILE

```
C SAMWRT  BIN  29NOV76  PROGRAM TO WRITE A SAM DATA FILE
C
C THE FILE IS 1000 WORDS LONG WRITTEN FROM ARRAY BUFF
C
C RESTRICTIONS: SAMFIL SHOULD NOT EXIST BEFORE RUNNING PROGRAM
C
C
      INTEGER*2 FUNIT1  /* FILE UNIT TO BE USED
      INTEGER*2 SAMFIL  /* FILE TYPE FOR SAM FILE
      INTEGER*2 BUFLNG  /* BUFFER LENGTH
C
      PARAMETER FUNIT1=1, SAMFIL=0, BUFLNG=1000
C
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 TYPE    /* CONTAINS FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ OR WRITTEN BY PRWF$$
      INTEGER*2 I
      INTEGER*2 CODE  /* HOLDS ERROR RETURN CODE
C
$INSERT SYSCOM>KEYS.F
C
C
C INITIALIZE BUFFER CONTENTS
     DO 10 I= 1, BUFLNG
       BUFF(I) = I
```

```
10    CONTINUE
C
C OPEN A NEW SAM DATA FILE CALLED 'SAMFIL' IN CURRENTLY ATTACHED
C UFD FOR WRITING ON FILE UNIT FUNIT1
C
C SINCE KEYS.F (KEY DEFINITIONS) DEFINES THE KEYS AS PARAMETERS
C THE USE OF MULTIPLE MNEMONIC KEYS WILL NOT GENERATE MORE CODE
C THAN THE USE OF NUMERIC KEYS. THE USE OF MNEMONIC KEYS IS
C RECOMMENDED AT ALL TIMES.
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$IUFD,'SAMFIL',6,FUNIT1,TYPE,
     X    CODE)
      IF (CODE.NE.0) GO TO 9010
      IF (TYPE .NE. SAMFIL) GO TO 9000   /* ERROR
C
C WRITE 1000 WORDS FROM BUFF INTO THE NEW DATA FILE
C
     CALL PRWF$$(K$WRIT,FUNIT1,LOC(BUFF),BUFLNG,INTL(0),NMREAD,
    X    CODE)
      IF (CODE.NE.0) GO TO 9010
C
C K$CLOS FILE. THIS RELEASES UNIT FUNIT1 FOR RE-USE AND INSURES
C ALL FILE BUFFERS HAVE BEEN WRITTEN TO DISK.
C NOTE PRIMOS WILL NOT AUTOMATICALLY K$CLOS FILES ON 'CALL EXIT'.
C
9000  CALL SRCH$$(K$CLOS, 0, 0, FUNIT1, 0, CODE)
      IF (CODE.NE.0) GO TO 9010
C
C RETURN TO PRIMOS
C
      CALL EXIT
      END
```

▶ WRITING A DAM FILE

```
C DAMWRT  BIN  29NOV76  PROGRAM TO WRITE A DAM DATA FILE
C
C NOTE THAT THE ONLY DIFFERENCE FROM PROGRAM SAMFIL IS THE
C 'NEW FILE' KEY SUPPLIED TO SRCH$$ IN CREATING THE FILE
C
C RESTRICTION: DAMFIL SHOULD NOT EXIST BEFORE RUNNING PROGRAM
C
C
      INTEGER*2 FUNIT1  /* FILE UNIT TO BE USED
      INTEGER*2 DAMFIL  /* FILE TYPE OF DAM DATA FILE
      INTEGER*2 BUFLNG  /* DATA BUFFER LENGTH IN WORDS
C
      PARAMETER FUNIT1=1, DAMFIL=1, BUFLNG=1000
C
```

```
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ OR WRITTEN BY PRWF$$
      INTEGER*2 CODE    /* ERROR CODE RETURNED FROM FILE SYSTEM
      INTEGER*2 I
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
C INITIALIZE BUFFER
C
      DO 1Ø I = 1, BUFLNG
        BUFF(I) = I
1Ø    CONTINUE
C
C INSURE THAT THE FILE 'DAMFIL' DOES NOT ALREADY EXIST
C
      CALL SRCH$$(K$READ+K$IUFD,'DAMFIL',6,FUNIT1,TYPE,CODE)
      IF (CODE .NE. E$FNTF) GO TO 9ØØØ  /* FILE ALREADY EXISTS
C
C OPEN A NEW DAM DATA FILE CALLED 'DAMFIL' IN THE CURRENT
C UFD FOR WRITING ON FILE UNIT FUNIT1 (I.E. CREATE NEW DAM FILE)
C
      CALL SRCH$$(K$WRIT+K$NDAM+K$IUFD,'DAMFIL',6,FUNIT1,TYPE,
     X   CODE)
      IF (CODE.NE.Ø) GO TO 9Ø1Ø
      IF (TYPE .NE. DAMFIL) STOP   /* WILL NEVER STOP
C
C WRITE THE BUFFER INTO THE FILE
C
     CALL PRWF$$(K$WRIT,FUNIT1,LOC(BUFF),BUFLNG,INTL(Ø),NMREAD,
     X   CODE)
     IF (CODE.NE.Ø) GO TO 9Ø1Ø
C
C K$CLOS THE FILE AND EXIT
C
9ØØØ  CALL SRCH$$(K$CLOS, Ø, Ø, FUNIT1, TYPE, CODE)
      IF (CODE.NE.Ø) GO TO 9Ø1Ø
      CALL EXIT
C
9Ø1Ø  CALL ERRPR$(K$NRTN,CODE,Ø,Ø,Ø,Ø)
      END
```

▶ READING A SAM OR DAM FILE

```
C REDFIL  BIN  29NOV76   READ SAM/DAM FILE, PRINT LARGEST INTEGER
C
C THIS PROGRAM SHOWS HOW TO USE THE 'CODE' ERROR RETURN
```

```
C MECHANISM AND SUBROUTINE ERRPR$ TO PRINT ERROR MESSAGES.
C
C NOTE THAT PROGRAM DOESN'T CHECK IF THE DATA FILE IS SAM OR DAM.
C TO USER'S PROGRAM, SAM OR DAM FILES ARE FUNCTIONALLY EQUIVALENT
C EXCEPT FOR ACCESS TIME TO RAMDOM POINTS IN THE FILE
C
C RESTRICTIONS: NONE
C
C
      INTEGER*2 FUNIT   /* FILE UNIT TO BE USED
      INTEGER*2 DAMFIL  /* TYPE OF DAM DATA FILE
      INTEGER*2 BUFLNG  /* LENGTH OF DATA BUFFER IN WORDS
C
      PARAMETER FUNIT=1, DAMFIL=2, BUFLNG=100
C
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ OR WRITTEN BY PRWF$$
      INTEGER*2 CODE    /* ERROR CODE RETURNED BY FILE SYSTEM
      INTEGER*2 LARGST  /* LARGEST UNSIGNED INTEGER IN FILE
      INTEGER*2 FNAME(16)  /* FILE NAME BUFFER
      INTEGER*2 I,N
C
      INTEGER*4 POSITN  /* 32BIT INTEGER POSITION FOR PRWF$$
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
C INITIALIZE AND GET FILE NAME FROM TERMINAL
C
      LARGST = -32767  /* LARGEST UNSIGNED INTEGER
10    WRITE(1,1000)  /* FORTRAN UNIT 1 IS TERMINAL
1000  FORMAT ('TYPE FILE NAME')
C
      READ(1,1010) (FNAME(I), I=1,16)
1010  FORMAT (16A2)
C
C OPEN FNAME IN CURRENTLY ATTACHED UFD FOR READING ON FILE UNIT 1
C (NOT THE SAME AS FORTRAN UNIT 1). CHECK FOR ERRORS.
C NOTE THAT THE NAME NEED NOT ACTUALLY BE 32 CHARACTERS LONG AS
C TRAILING BLANKS ARE IGNORED.
C
      CALL SRCH$$(K$READ+K$IUFD,FNAME,32,FUNIT,TYPE,CODE)
      IF (CODE .EQ. 0) GO TO 100  /* NO ERRORS
C
C PRINT THE SYSTEM ERROR MSG AND IMMEDIATELY RTRN TO THIS PROGRAM
C IF THE ERROR IS 'FILE NOT FOUND', GET ANOTHER NAME.
C GIVE UP ON ALL OTHER ERRORS
C
      CALL ERRPR$(K$IRTN, CODE, FNAME, 32, 'REDFIL', 6)
      IF (CODE.EQ.E$FNTF) GO TO 10 /*NOT FOUND-GET ANOTHER NAME
      GO TO 9010   /* ANOTHER TYPE OF ERROR - GIVE UP
```

```
C
C THE FILE HAS BEEN OPENED.
C MAKE SURE THE FILE IS NOT A DIRECTORY
C
100   IF (TYPE .GT. DAMFIL) GO TO 9000   /* IS A DIRECTORY
C
C READ AN 'OPTIMAL' NUMBER OF WORDS UP TO BUFLNG WORDS FROM FILE.
C SET LARGST TO THE LARGEST UNSIGNED INTEGER IN THE FILE.
C CHECK FOR END-OF-FILE.
C
30    CALL PRWF$$(K$READ+K$CONV, FUNIT, LOC(BUFF),BUFLNG,
     X                                  INTL(0),NMREAD,CODE)
      IF (CODE .EQ. E$EOF) GO TO 31  /* END-OF-FILE
      IF (CODE .NE. 0) GO TO 9010  /* SOME OTHER ERROR
31    DO 40 I= 1, NMREAD   /* FOR EACH WORD ACTUALLY READ
         IF ((LARGST.LE.0).AND.(BUFF(I).GE.0)) LARGST = BUFF(I)
         IF (LARGST .LT. BUFF(I)) LARGST = BUFF(I)
40    CONTINUE
      IF (CODE .NE. E$EOF) GO TO 30  /* MORE DATA IN FILE
C
C FIND OUT IF THE DATA FILE IS EMPTY
C GET CURRENT FILE POINTER POSITION WHICH IS NOW AT END-OF-FILE.
C IF THE POSITION IS 0, THE FILE IS EMPTY
C
      CALL PRWF$$(K$RPOS, FUNIT, 0, 0, POSITN, NMREAD, CODE)
      IF (CODE .NE. 0) GO TO 9010  /* ERROR
      IF (POSITN .GT. 0) GO TO 50  /* NOT A NULL FILE
      WRITE(1,1030)
1030  FORMAT ('FILE EMPTY')
      GO TO 9000   /* EXIT
C
C FILE NOT EMPTY. PRINT LARGEST INTEGER
C
50    WRITE(1,1020) LARGST
1020  FORMAT ('LARGEST INTEGER IN FILE IS ',I6)
      GO TO 9000  /* EXIT
C
C K$CLOS FILES   EXIT
C PRINT ERROR MESSAGE IF NECESSARY
C
9010  CALL ERRPR$(K$IRTN, CODE, 0, 0, 'REDFIL', 6)
C
9000  CALL SRCH$$(K$CLOS, 0, 0, FUNIT, TYPE, CODE)
      IF (CODE.NE.0) GO TO 9010
      CALL EXIT
      END
```

▶ CREATING A SEGMENT DIRECTORY

```
C CRTSEG  BIN  29NOV76  CREATE A SEGMENT DIRECTORY
C                AND WRITE DATA FILE IN IT
C
C RESTRICTIONS: SEGDIR SHOULD NOT EXIST BEFORE RUNNING PROGRAM
C
C
      INTEGER*2 BUFLNG  /* DATA BUFFER LENGTH
      INTEGER*2 SAMSEG  /* FILE TYPE OF SAM SEGMENT DIRECTORY
      INTEGER*2 SGUNIT  /* FILE UNIT FOR SEGMENT DIRECTORY
      INTEGER*2 FUNIT   /* FILE UNIT FOR DATA FILE
C
      PARAMETER BUFLNG=1Ø, SAMSEG=2, SGUNIT=1, FUNIT=2
C
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ OR WRITTEN BY PRWF$$
      INTEGER*2 I
      INTEGER*2 CODE   /* RETURN CODE STORED HERE
      INTEGER*2 CODEA  /* SCRATCH CODE
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
C INITIALIZE DATA BUFFER CONTENTS
C
      DO 1Ø I= 1, BUFLNG
        BUFF(I) = I
1Ø    CONTINUE
C
C OPEN A NEW SAM SEGMENT DIRECTORY CA LED 'SAMDIR' IN CURRENTLY
C ATTACHED UFD FOR READING AND WRITINC ON FILE UNIT SGUNIT.
C NOTE: SEGDIRS OPEN FOR WRITE ONLY WI L NOT BE HANDLED CORRECTLY
C
      CALL SRCH$$(K$RDWR+K$NSGS+K$IUFD,'SEGDIR',6,SGUNIT,TYPE,
     X    CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
      IF (TYPE.NE.SAMSEG) GO TO 95ØØ  /* ERROR--MUST HAVE EXISTED
C
C ENTER A NEW SAM DATA FILE (I.E. OPEN SAM DATA FILE FOR WRITING)
C IN THE JUST CREATED SEGMENT DIRECTORY. THE NEW DATA FILE
C WILL BE ENTRY Ø IN THE SEGMENT DIRECTORY.
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,SGUNIT,Ø,FUNIT,TYPE,CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
C
C WRITE THE DATA BUFFER INTO THE JUST CREATED SAM FILE.
C K$CLOS THE DATA FILE.
C
```

```
      CALL PRWF$$(K$WRIT,FUNIT,LOC(BUFF),BUFLNG,INTL(Ø),NMREAD,
     X    CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
      CALL SRCH$$(K$CLOS, Ø, Ø, FUNIT, Ø, CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
C
C REPLACE BUFF WITH NEW DATA
C
      DO 2Ø I= 1, BUFLNG
        BUFF(I) = I * 1Ø
2Ø    CONTINUE
C
C OPEN A DIFFERENT NEW SAM DATA FILE ON FUNIT FOR WRITING
C (I.E. ENTER ANOTHER FILE IN SEGMENT DIRECTORY). THIS IS DONE
C IN TWO STEPS. FIRST THE FILE POINTER OF THE SEGMENT DIR UNIT IS
C POSITIONED TO THE ENTRY NUMBER DESIRED. THE SRCH$$ IS
C CALLED AS ABOVE.
C
      CALL SGDR$$(K$SPOS,SGUNIT, 1, I, CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
      IF (I .NE. -1) GO TO 95ØØ  /* ERROR EXIT
C
C NOTE THAT THE SEGMENT DIRECTORY OPEN ON SGUNIT HAS ONLY 1 ENTRY
C (ENTRY Ø) AT THIS TIME. THUS, POSITIONING TO ENTRY 1
C WILL POSITION TO END-OF-FILE (NOT BEYOND) AND THE FOLLOWING
C CALL TO SRCH$$ WILL CAUSE THE SEGMENT DIRECTORY TO BE EXTENDED
C IN LENGTH BY ONE ENTRY.
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,SGUNIT,Ø,FUNIT,TYPE,CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
C
C WRITE DATA INTO THE SAM FILE   THE K$CLOS THE FILE
C
      CALL PRWF$$(K$WRIT,FUNIT,LOC(BUFF),BUFLNG,INTL(Ø),NMREAD,
     X    CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
      CALL SRCH$$(K$CLOS, Ø, Ø, FUNIT, Ø, CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
C
C REPLACE THE BUFFER WITH NEW DATA
C
      DO 3Ø I= 1, BUFLNG
        BUFF(I) = I * 1ØØ
3Ø    CONTINUE
C
C MAKE THE SEGMENT DIRECTORY ITSELF LARGE ENOUGH TO CONTAIN
C 1Ø ENTRIES. PLACE A SAM FILE IN THE 1ØTH ENTRY.
C
      CALL SGDR$$(K$MSIZ, SGUNIT, 1Ø, Ø, CODE)
      IF (CODE.NE.Ø) GO TO 95ØØ
C
C THE FILE POINTER ASSOCIATED WITH SGUNIT IS NOW AT END-OF-FILE.
C A CALL TO SRCH$$ WITHOUT FURTHER POSITIONING THE SEGMENT
```

```
C DIRECTORY'S FILE POINTER WOULD EXTEND THE SEGMENT DIRECTORY
C AND ENTER THE NEW FILE AS TH 11TH ENTRY. THEREFORE, SGDR$$
C MUST BE CALLED TO POSITION TO THE 10TH ENTRY.
C
      CALL SGDR$$(K$SPOS, SGUNIT, 10, I, CODE)
      IF (CODE.NE.0) GO TO 9500
      IF (I .NE. 0) STOP  /* FILE CANNOT BE PRESENT
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,SGUNIT,0,FUNIT,TYPE,CODE)
      IF (CODE.NE.0) GO TO 9500
      CALL PRWF$$(K$WRIT,FUNIT,LOC(BUFF),BUFLNG,INTL(0),NMREAD,
     X    CODE)
      IF (CODE.NE.0) GO TO 9500
      CALL SRCH$$(K$CLOS, 0, 0, FUNIT, TYPE, CODE)
      IF (CODE.NE.0) GO TO 9500
C
C K$CLOS SEGMENT DIRECTORY   EXIT
C
      CALL SRCH$$(K$CLOS, 0, 0, SGUNIT, TYPE, CODE)
      IF (CODE.NE.0) GO TO 9500
      CALL EXIT
C
C ERROR EXIT. K$CLOS ALL UNITS. PRINT ERROR MESSAGE AND DO NOT
C ALLOW RESTART. E$NULL IS THE NULL SYSTEM ERROR, I.E.,
C NO SYSTEM ERROR MESSAGE IS PRINTED.
C
9500  CALL SRCH$$(K$CLOS, 0, 0, FUNIT, TYPE, CODEA)
      CALL SRCH$$(K$CLOS, 0, 0, SGUNIT, TYPE, CODEA)
      CALL ERRPR$(K$NRTN,CODE,'UNEXPECTED ERROR',16,'CRTSEG',6)
C
      END
```

▶ READING A LOGICAL RECORD FROM A FILE

```
C RDLREC  BIN  29NOV76  READ A LOGICAL RECORD FROM A FILE
C
C PROGRAM READS LOGICAL RECORD 'N' FROM A FILE CONSISTING
C OF FIXED LENGTH RECORDS
C
C IN THIS PROGRAM, THE FILE ACCESSED IS CONSIDERED TO CONTAIN AN
C UNLIMITED NUMBER OF LOGICAL RECORDS. EACH RECORD CONTAINS 'M'
C WORDS. THE PROGRAM READS AND PRINTS TO THE TERMINAL THE
C CONTENTS OF RECORD NUMBER N AS M INTEGERS. THE FIRST RECORD
C OF A FILE IS RECORD NUMBER 0 (ZERO).
C NOTE THAT A LOGICAL RECORD IS MERELY A GROUPING OF WORDS IN A
C FILE. THE LOGICAL RECORD SIZE HAS NO RELATION TO THE PHYSICAL
C RECORD SIZE OF THE DISK.
C
C RESTRICTIONS:
```

```
C   1. RECORD SIZE MUST BE BETWEEN 1 AND BUFFER LENGTH
C   2. RECORD NUMBER MUST BE BETWEEN Ø AND 32767
C   3. THE RECORD MUST BE IN THE FILE
C   4. THE FILE MUST PREVIOUSLY EXIST
C   5. THE FILE MUST BE A DATA FILE (SAMFIL OR DAMFIL)
C
C
      INTEGER*2 FUNIT1  /* PRIMOS FILE UNIT USED FOR DATA FILE
      INTEGER*2 BUFLNG  /* LENGTH OF DATA BUFFER
C
      PARAMETER FUNIT1=1, BUFLNG=1ØØØ
C
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 FNAME(16)  /* FILE NAME BUFFER
      INTEGER*2 RECSIZ  /* NUMBER WORDS IN A LOGICAL RECORD
      INTEGER*2 RECNUM  /* LOGICAL RECORD NUMBER
      INTEGER*2  TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ, RETURNED BY PRWF$$
      INTEGER*2 CODE    /* ERROR STATUS RETURNED BY FILE SYSTEM
      INTEGER*2 I
C
      INTEGER*4 POSITN  /* 32BIT WORD NR USED AS POS TO PRWF$$
C
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
C ASK FOR FILE NAME
C
1Ø    WRITE(1,1ØØØ)   /* FORTRAN UNIT 1 IS TTY
1ØØØ  FORMAT ('TYPE FILE NAME')
C
C READ FILE NAME
C
      READ(1,1Ø1Ø) (FNAME(I),I=1,16)
1Ø1Ø  FORMAT (16A2)
C
C OPEN FNAME IN CURRENT UFD FOR READING ON FILE UNIT FUNIT1
C
      CALL SRCH$$(K$READ+K$IUFD, FNAME, 32, FUNIT1, TYPE, CODE)
     / IF (CODE.NE.Ø) GO TO 1ØØØ
C
C ASK FOR LOGICAL RECORD SIZE
C
2Ø    WRITE(1,1Ø2Ø)
1Ø2Ø  FORMAT ('TYPE RECORD SIZE')
      READ(1,1Ø3Ø) RECSIZ
1Ø3Ø  FORMAT (I6)
      IF (RECSIZ .GE. 1 .AND. RECSIZ .LE. BUFLNG) GO TO 3Ø
      WRITE(1,1Ø4Ø)
1Ø4Ø  FORMAT ('BAD RECORD SIZE')
      GO TO 2Ø
```

```
C
C ASK FOR RECORD NUMBER. FIRST RECORD IS NUMBERED Ø (ZERO)
C
3Ø    WRITE(1,1Ø5Ø)
1Ø5Ø  FORMAT ('TYPE RECORD NUMBER')
      READ (1,1Ø3Ø) RECNUM
      IF (RECNUM .GE. Ø) GO TO 35
      WRITE(1,1Ø51)
1Ø51  FORMAT ('BAD RECORD NUMBER')
      GO TO 3Ø
C
C CALCULATE THE 32-BIT WORD NUMBER OF THE FIRST WORD IN THE
C DESIRED RECORD. NOTE THAT IF BOTH RECSIZ AND RECNUM ARE BOTH
C POSITIVE 16BIT NUMBERS, THE 32BIT WORD NUMBER MUST ALSO BE
C POSITIVE.
C
C POSITIONING MAY BE DONE TO AN ABSOLUTE WORD NUMBER OR RELATIVE
C TO THE CURRENT POSITION. SINCE A JUST OPENED FILE IS ALWAYS
C POSITIONED TO TOP-OF-FILE AND THE CALCULATED WORD NUMBER WILL
C NEVER BE NEGATIVE, THE ARGUMENT FOR POSITION TO PRWF$$ WILL
C BE THE SAME FOR BOTH CALLS IN THIS PROGRAM.
C
35    POSITN=INTL(RECSIZ)*INTL(RECNUM)    /* POSITN IS INTEGER*4
      IF (POSITN .GT. 32767) GO TO 1ØØ  /* ABSOLUTE POSITIONING
C
C RECORD LESS THAN 32767 WORDS FROM THE BEGINNING, USE RELATIVE
C POSITIONING.
C NOTE THAT ABSOLUTE POSITIONING COULD HAVE BEEN USED FOR A
C RECORD ANYWHERE IN THE FILE, NOT JUST FOR THOSE RECORDS
C BEYOND WORD 32767. RELATIVE IS SHOWN HERE ONLY FOR EXAMPLE.
C
C NOTE ALSO THAT RELATIVE POSITIONING COULD BE USED TO POSITION
C TO ANY WORD IN THE FILE, GIVEN THE RESTICTIONS ON RECSIZ AND
C RECNUM.
C
C WHEN REL POSITIONING IS USED, THE POS ARGUMENT (POSITN HERE)
C IS CONSIDERED TO BE A SIGNED 32-BIT INTEGER.
C
      CALL PRWF$$(K$READ+K$PRER,FUNIT1,LOC(BUFF),RECSIZ,POSITN,
     X                                          NMREAD, CODE)
      GO TO 2ØØ  /* SKIP OVER ABSOLUTE POSITION EXAMPLE
C
C RECORD IS MORE THAN 32767 WORDS FROM THE BEGINNING OF FILE, USE
C ABSOLUTE POSITIONING.
C
C WHEN ABSOLUTE POSITIONING IS USED, POSITION ARGUMENT (POSITN)
C IS CONSIDERED TO BE AN SIGNED 32-BIT INTEGER.
C NOTE THAT THE E$BOF ERROR (BEGINNING OF FILE) CAN OCCUR.
C
1ØØ   CALL PRWF$$(K$READ+K$PREA,FUNIT1,LOC(BUFF),RECSIZ,POSITN,
     X                                          NMREAD, CODE)
C
2ØØ   IF (CODE .NE. Ø) GO TO 3ØØ  /* ERROR DETECTED
```

```
C
C HAVE READ RECORD, NOW TYPE IT.
C
      WRITE(1,1060) RECNUM,RECSIZ
1060  FORMAT('RECORD ',I6,' CONTAINS ',I6,' ENTRIES AS FOLLOWS')
      WRITE(1,1070)  (BUFF(I), I=1,RECSIZ)
1070  FORMAT (10I7)
C
C RETURN TO PRIMOS AFTER CLOSING THE FILE
C
250   CALL SRCH$$(K$CLOS, 0, 0, FUNIT1, TYPE, CODE)
      IF (CODE.NE.0) GO TO 1000
      CALL EXIT
      GO TO 10  /* START COMMAND RESTARTS PROGRAM
C
C ERROR WHILE ATTEMPTING TO READ THE RECORD
C
300   CALL ERRPR$(K$IRTN, CODE, 0, 0, 'RDLREC', 6)
      IF (CODE .NE. E$EOF) GO TO 250  /* EXIT IF NOT END-OF-FILE
C
C END-OF-FILE REACHED.
C REWIND FILE AND TRY AGAIN
C
      CALL PRWF$$(K$POSN+K$PREA,FUNIT1,0,0,INTL(0),NMREAD,
     X    CODE)
      IF (CODE.NE.0) GO TO 1000
      GO TO 20
C
1000  CALL ERRPR$(K$NRTN,CODE,0,0,0,0)
      END
```

▶ READING A FILE IN A SEGMENT DIRECTORY

```
C REDSEG  BIN  29NOV76  READ FILE IN A SEGMENT DIRECTORY
C
C THIS PROGRAM READS FILE NUMBER N IN  SEGMENT DIRECTORY AND
C TYPES WORD NUMBER M IN THAT FILE. THE FIRST FILE IN THE
C DIRECTORY IS FILE NUMBER Ø. THE FIRST WORD IN THE FILE IS
C WORD NUMBER Ø.
C
C RESTRICTIONS:
C   1. THE SEGMENT DIRECTORY FILE MUST EXIST
C   2. THE FILE NUMBER MUST BE BETWEEN Ø AND 32767
C   3. THE FILE MUST BE IN THE SEGMENT DIRECTORY
C   4. THE WORD NUMBER MUST BE BETWEEN Ø AND 32767
C   5. THE WORD MUST BE IN THE FILE.
C
C
      INTEGER*2 FUNIT   /* PRIMOS FILE UNIT FOR DATA FILE
      INTEGER*2 SGUNIT  /* PRIMOS FILE UNIT FOR SEGMENT DIRECTORY
      INTEGER*2 SAMSEG  /* FILE TYPE OF SAM SEGMENT DIRECTORY
      INTEGER*2 DAMSEG  /* FILE TYPE OF DAM SEGMENT DIRECTORY
C
      PARAMETER FUNIT=2, SGUNIT=1, SAMSEG=2, DAMSEG=3
C
      INTEGER*2 BUFF    /* DATA BUFFER
      INTEGER*2 SEGDIR(16)  /* NAME OF SEGMENT DIRECTORY BUFFER
      INTEGER*2 FILNUM  /* FILE NR (ENTRY NR) OF FILE IN SEGDIR
      INTEGER*2 WRDNUM  /* WORD NUMBER IN DATA FILE TO BE READ
      INTEGER*2 CODE    /* ERROR CODE RETURNED BY FILE SYSTEM
      INTEGER*2 TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NR WORDS READ/WRITTEN/RTRNED BY PRWF$$
      INTEGER*2 I
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
C INSURE FILE UNITS TO BE USED ARE K$CLOSD
C ASK FOR AND READ SEGMENT DIRECTORY NAME FROM TERMINAL
C
10    CALL SRCH$$(K$CLOS, Ø, Ø, SGUNIT, Ø, CODE)
      IF (CODE.NE.Ø) GO TO 1ØØ
      CALL SRCH$$(K$CLOS, Ø, Ø, FUNIT, Ø, CODE)
      IF (CODE.NE.Ø) GO TO 1ØØ
      WRITE(1,1ØØØ)
1ØØØ  FORMAT ('TYPE SEGMENT DIRECTORY NAME')
      READ (1,1Ø1Ø) (SEGDIR(I), I=1,16)
1Ø1Ø  FORMAT (16A2)
```

```
C
C OPEN THE SEGMENT DIRECTORY FOR READING ON SGUNIT
C
      CALL SRCH$$(K$READ+K$IUFD, 'SEGDIR', 6, SGUNIT, TYPE, CODE)
      IF (CODE.NE.Ø) GO TO 1ØØ
C
C TYPE CONTAINS THE FILE TYPE OF THE FILE JUST OPENED.
C MAKE SURE THE FILE IS EITHER A SAM OR DAM SEGMENT DIRECTORY.
C ALLOWABLE TYPE VALUES ARE 2 AND 3.
C
      IF (TYPE .EQ. SAMSEG) GO TO 2Ø
      IF (TYPE .EQ. DAMSEG) GO TO 2Ø
C
C NOT A SEGMENT DIRECTORY - TRY AGAIN
C
      WRITE(1,1Ø2Ø)
1Ø2Ø  FORMAT('FILE IS NOT A SEGMENT DIRECTORY')
      GO TO 1Ø
C
C ASK FOR FILE (ENTRY) NUMBER IN SEGMENT DIRECTORY
C
2Ø    WRITE(1,1Ø3Ø)
1Ø3Ø  FORMAT ('TYPE FILE NUMBER')
      READ (1,1Ø4Ø) FILNUM
1Ø4Ø  FORMAT (I6)
      IF (FILNUM .LT. Ø) GO TO 2Ø
C
C ASK FOR WORD NUMBER IN DATA FILE TO READ
C
3Ø    WRITE(1,1Ø35)
1Ø35  FORMAT ('TYPE WORD NUMBER')
      READ (1,1Ø4Ø) WRDNUM
      IF (WRDNUM .LT. Ø) GO TO 3Ø
C
C TRY TO POSITION TO WORD NUMBER IN THE SEGMENT DIRECTORY.
C IF END-OF-FILE REACHED, FILE IS NOT IN SEGMENT DIRECTORY.
C SGDR$$ RETURNS THE VALUE 1 IN THE 4TH ARGUMENT (TYPE) IF A
C FILE IS ENTERED IN THE ENTRY POSITION. THIS PROGRAM DOES NOT
C CHECK THE VALUE, SINCE SRCH$$ WILL RETURN THE PROPER ERROR CODE
C (E$FNTS - FILE NOT FOUND IN SEGMENT DIRECTORY) ANYHOW.
C
      CALL SGDR$$(K$SPOS, SGUNIT, FILNUM, TYPE, CODE)
      IF (CODE .EQ. E$EOF) CODE = E$FNTS  /* FILE NOT FOUND
      IF (CODE .NE. Ø) GO TO 1ØØ
C
C OPEN FILE IN SEGMENT DIRECTORY FOR READING
C
      CALL SRCH$$(K$READ+K$ISEG,SGUNIT,Ø,FUNIT,TYPE,CODE)
      IF (CODE .NE. Ø) GO TO 1ØØ
C
C PRINT THE WORD, K$CLOS THE FILES, AND RETURN TO PRIMOS
C
      WRITE(1,1Ø5Ø) WRDNUM,FILNUM,(SEGDIR(I), I= 1,16),BUFF
```

```
105Ø  FORMAT ('WORD',I6,' OF FILE (',I6,') IN ',16A2,
     X  'CONTAINS',I6)
5Ø    CALL SRCH$$(K$CLOS, Ø, Ø, FUNIT, Ø, CODE)
      CALL SRCH$$(K$CLOS, Ø, Ø, SGUNIT, Ø, CODE)
      CALL EXIT
      GO TO 1Ø  /* START COMMAND RE-STARTS PROGRAM
C
C
C COMMON ERROR HANDLER
C NOTE THAT THE NEW FILE SYS PROPERLY DIFFERENTIATES THE VARIOUS
C ERRORS WHICH FORMERLY WERE GROUPED UNDER OLD ERROR CODE 'SQ'
C
1ØØ   IF (CODE.EQ.E$FNTS) GO TO 11Ø /* FILE NOT FOUND IN SEGDIR
      IF (CODE .EQ. E$EOF ) GO TO 12Ø   /* END-OF-FILE
      CALL ERRPR$(K$IRTN,CODE,Ø,Ø,'REDSEG',6) /* PRINT ERROR MSG
      GO TO 5Ø   /* K$CLOS FILES   EXIT
C
C FILE NOT FOUND IN SEGMENT DIRECTORY
C LET THE USER TRY AGAIN
C
11Ø   WRITE(1,1Ø6Ø) FILNUM, (SEGDIR(I), I=1, 16)
1Ø6Ø  FORMAT ('FILE (',I6,') NOT FOUND IN ',16A2)
      GO TO 1Ø  /* RE-TRY
C
C END-OF-FILE
C CODE WILL CONTAIN E$EOF ONLY WHILE TRYING TO READ
C THE DATA FILE. ALLOW RE-TRY.
C
12Ø   WRITE(1,1Ø7Ø) WRDNUM,FILNUM,(SEGDIR(I),I=1,16)
1Ø7Ø  FORMAT ('WORD',I6,' NOT IN FILE (',I6,') IN ',16A2)
      GO TO 1Ø  /* RE-TRY
C
      END
```

Part III
Math and Application Library Subroutines

SECTION 7

FORTRAN STANDARD FUNCTIONS

INTRODUCTION

The subroutines described in this section are Prime subroutines that correspond to those defined as ANSI-standard FORTRAN functions. Table 7-1 describes these functions. They are all called using the standard FORTRAN calling sequences for R or V mode functions.

FUNCTION REFERENCES

Library function references are of the form:

R = name (argument-1,...,argument-n)

where R is a variable defined within the scope of the program, name is one of the library function names and argument-1,...,argument-n is a list of arguments to be processed by the function. Most functions require only one argument; e.g., A=SIN(X). See the FORTRAN Programmers Guide for a discussion of function references and for examples of their use.

Fixed Point Data Storage

Fixed point data is stored in the A-register (single precision). The A-register may be referred to as the fixed point accumulator.

Extended Registers

Locations AC1, AC2, AC3 and AC4 provide accumulators for complex subroutines. These are global symbols defined in the FORTRAN library. AC1 and AC2 are for the real part and AC3 and AC4 for the imaginary part.

The global symbol AC5 contains error conditions which may be generated by FORTRAN library subroutines.

SINGLE ARGUMENT SCIENTIFIC FUNCTIONS

The "$X" series are short callable (V-mode only) versions of common scientific functions which take a single argument in the single or double precision floating accumulator. The routines available are:

ALOG$X
ATAN$X
COS$X
DATN$X
DCOS$X
DEXP$X
DL1Ø$X
DLOG$X
DLG2$X
DSIN$X
DSQR$X
EXP$X
SIN$X
SQRT$X

A FORTRAN user need not call these functions explicitly; the FORTRAN compiler generates calls to them in response to normal function usage.

FORTRAN 77 FUNCTIONS

FORTRAN 77 applies all of the functions in Table 7-1 plus those listed in Table 7-2. See the FORTRAN 77 Reference Guide (IDR4Ø29) for more details.

Table 7-1. FORTRAN Library Functions

Function	Subroutine	Operation	Number of Arguments	Type of Argument	Type of Result
Absolute value:					
- Real	ABS	|arg|	1	Real	Real
- Integer	IABS(1)	|arg|	1	Integer	Integer
- Double precision	DABS	|arg|	1	Double	Double
- Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real
Conversion:					
- Integer to real	FLOAT		1	Integer	REAL
- Real to integer	IFIX	Result is largest integer $\leq a$	1	Real	Integer
- Double to real	SNGL		1	Double	Real
- Real to double	DBLE		1	Real	Double
- Complex to real (obtain real part)	REAL		1	Complex	Real
- Complex to real (obtain imaginary - part)	AIMAG		1	Complex	Real
- Real to complex	CMPLX	$c=Arg_1+i*Arg_2$	2	Real	Complex
Truncation:					
- Real to real	AINT	sign of arg* largest integer $\leq$ arg	1	Real	Real
- Real to integer	INT		1	Real	Integer
- Double to integer	IDINT		1	Double	Integer
- Double to double	DINT		1	Double	Double
Remaindering:					
- Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real
- Integer	MOD(1)		2	Integer	Integer
- Double precision	DMOD		2	Double	Double
Maximum Value:					
	AMAX0	$Max(Arg_1,Arg_2,...)$	2,3,or 4	Integer	Real
	AMAX1			Real	Real
	MAX0			Integer	Integer
	MAX1			Real	Integer
	DMAX1			Double	Double
Minimum Value:					
	AMIN0	$MIN(Arg_1,Arg_2,...)$	2,3,or 4	Integer	Real
	AMIN1			Real	Real
	MIN0			Integer	Integer
	MIN1			Real	INTEGER
	DMIN1			Double	Double

Table 7-1. (continued)

Function	Subroutine	Operation	Number of Arguments	Type of Argument	Type of Result
Transfer of Sign:					
- Real	SIGN	$\{Sgn(Arg_2) * \lvert Arg_1 \rvert\}$	2	Real	Real
- Integer	ISIGN		2	Integer	Integer
- Double Precision	DSIGN		2	Double	Double
Positive Difference:					
- Real	DIM	$\{Arg_1 - Min(Arg_1, Arg_2)\}$	2	Real	Real
- Integer	IDIM		2	Integer	Integer
Exponential:					
- Real	EXP	$\{e^{Arg}\}$	1	Real	Real
- Double	DEXP		1	Double	Double
- Complex	CEXP		1	Complex	Complex
Logarithm:					
- Real	ALOG	log (Arg)	1	Real	Real
	ALOG10	$\log_{10}$ (Arg)	1	Real	Real
- Double	DLOG	$\log_e$ (Arg)	1	Double	Double
	DLOG2	$\log_2$ (Arg)	1	Double	Double
	DLOG10	$\log_{10}$ (Arg)	1	Double	Double
- Complex	CLOG	$\log_e$ (Arg)	1	Complex	Complex
Square Root:					
- Real	SQRT	$(Arg)^{1/2}$	1	Real	Real
- Double	DSQRT	$(Arg)^{1/2}$	1	Double	Double
- Complex	CSQRT	$c=(x+iy)^{1/2}$	1	Complex	Complex
Sine: (radians)					
- Real	SIN	$\{\sin (Arg)\}$	1	Real	Real
- Double	DSIN (2)		1	Double	Double
- Complex	CSIN		1	Complex	Complex
Cosine: (radians)					
- Real	COS	$\{\cos (Arg)\}$	1	Real	Real
- Double	DCOS (2)		1	Double	Double
- Complex	CCOS		1	Complex	Complex

Table 7-1. (continued)

Function	Subroutine	Operation	Number of Arguments	Type of Argument	Result
Hyperbolic - Tangent	TANH	tanh(Arg)	1	Real	Real
Arc Tangent:					
- Real	ATAN	arctan(Arg)	1	Real	Real
- Double	DATAN	arctan(Arg)	1	Double	Double
- quotient of two arguments	ATAN2	$\arctan(Arg_1/Arg_2)$	2	Real	Real
	DATAN2	$\arctan(Arg_1/Arg_2)$	2	Double	Double
Complex Conjugate	CONJG	Arg=X+iY, CONJG=X-iY	1	Complex	Complex
Random Number: (3)					
	RND	pick a random number from Ø to 1.Ø.	1	Integer	Real
	IRND	pick a random number from 0 to 32767 $(2^{15}-1)$	1	Integer	Integer

Notes

(1) These functions are not in the library but are expanded in line by FORTRAN compiler.

(2) Ranges for DSIN and DCOS:

DSIN(X)
X< 3.37E9 For V-mode
X< 1.69E9 For R-mode

DCOS(X)
X< 3.37E9 For V-mode
X< 1.69E9 For R-mode

(3) The argument for RND and IRND is interpreted as follows:

Arg>Ø, Arg is used to initialize the random number generator. Arg is returned as the value the call.

Arg=Ø, The function returns a random number: from Ø to 1.0 for RND, 0 to 32767 for IRND.

Arg<Ø, Initializes the random number generator and then returns a random number as in the Arg=Ø case.

Table 7-2. FORTRAN 77 Additional Functions

Function	Subroutine	Number of Arguments	Type of Argument	Result
Numeric to Double Precision	DREAL	1	Complex*16	Double
Numeric to Complex*16	DCMPLX	1 or 2	Integer	Complex*16
			Real	Complex*16
			Double	Complex*16
			Complex	Complex*16
			Complex*16	Complex*16
Character to Integer	ICHAR	1	Character	Integer
Integer to Character	CHAR	1	Integer	Character
Nearest whole Number	ANINT	1	Real	Real
	DINT	1	Double	Double
Nearest Integer	NINT	1	Real	Integer
	IDNINT	1	Double	Integer
Absolute value	CDABS	1	Complex*16	Double
Positive Difference	DDIM	2	Double	Double
Double Precision Product	DPROD	2	Real	Double
Length of Character Entity	LEN	1	Character	Integer
Index of a Substring	INDEX	2	Character	Integer
Real Part of Complex Argument	DREAL	1	Complex*16	Double
Imaginary Part of Complex Argument	AIMAG	1	Complex	Real
	DIMAG	1	Complex*16	Double
Conjugate of a Complex Argument	CONJ	1	Complex	Complex
	DCONJ	1	Complex*16	Complex
Square root	CDSQRT	1	Complex*16	Complex*16

Table 7-2. (continued)

Function	Subroutine	Number of Arguments	Type of Argument	Result
Exponential	CDEXP	1	Complex*16	Complex*16
Natural Logarithm	CDLOG	1	Complex*16	Complex*16
Sine	CDSIN	1	Complex*16	Complex*16
Cosine	CDCOS	1	Complex*16	Complex*16
Tangent	TAN	1	Real	Real
	DTAN	1	Double	Double
Arcsine	ASIN	1	Real	Real
	DASIN	1	Double	Double
Arccosine	ACOS	1	Real	Real
	DACOS	1	Double	Double
Hyperbolic Sine	SINH	1	Real	Real
	DSINH	1	Double	Double
Hyperbolic Cosine	COSH	1	Real	Real
	DCOSH	1	Double	Double
Hyperbolic Tangent	DTANH	1	Double	Double
Lexically Greater Than or Equal	LGE	2	Character	Logical
Lexically Greater Than	LGT	2	Character	Logical
Lexically Less Than or Equal	LLE	2	Character	Logical
Lexically Less Than	LLT	2	Character	Logical

SECTION 8

LOGICAL FUNCTIONS

This section describes FORTRAN logical functions which are not Library subroutines, but are expanded by the compiler. They are included in this document for reference purposes. These functions accept long integer as well as short integer arguments. The result of a mixed mode AND, OR, or XOR is long integer. The short integer argument is converted to long integer. Note that the conversion sign extends so that if bit 1 of the short integer is 1, bits 1-17 will be 1 after conversion to long integers; the result of a mixed mode shift or truncate is the mode of its first argument.

▶ AND

Performs a logical AND operation, bit by bit, on a variable list of integers.

```
i = AND (i1, i2, ..., in)
```

▶ LS

Shifts an integer variable left by a specified number of bits; vacated bits are filled with zeroes.

```
i2 = LS (i1, ip)
```

ip is the number of bits on i2 to be shifted to the left. If ip≤Ø, no change is made to the integer.

▶ LT

Preserves a specified number of left-most bits and sets the reset to zero (left truncation).

```
i2 = LT (i1, ip)
```

The first ip bits are set from the left are saved and the rest of the bits are set to zero. If ip ≤Ø, the entire integer is set to zero.

▶ OR

Performs a logical (inclusive) OR operation on a variable list of integers.

```
i = OR (i1, i2, ..., in)
```

▶ RS

Shifts an integer variable right by a specified number of bits; vacated bits are filled with zeroes.

```
i2 = RS (i1, ip)
```

ip is the number of bits to be shifted to the right on Integer "i1". If ip $\leq$ Ø, no change is made to the integer.

▶ RT

Preserves a specified number of right-most bits and sets the rest to zero (right truncation).

```
i2 = RT (i1, ip)
```

The first ip bits of i1 from the right are saved and the rest of the bits are set to zero. If ip$\leq$Ø, the entire integer is set to zero.

▶ SHFT

SHFT performs logical shift operations on integer variables. Format 1:

```
is = SHFT (i,ip1)
```

performs a shift operation on the variable. If ip1>Ø, the shift is to the right; if ip1<Ø the shift is to the left. If ip1=Ø, no shift occurs. This operation is equivalent to the RS function, and is provided for compatibility with other FORTRAN compilers. Format 2:

```
is = SHFT (i, ip1, ip2)
```

performs two shift operations, first by ip1 (setting zeroes in vacated bits), then by ip2 (setting zeroes in vacated bits). The sign of ip1 and ip2 determine the direction of the shift while their magnitude determines the number of bits to be shifted.

▶ XOR

Performs a logical exclusive OR on a variable list of integers.

```
i = XOR (i1, i2, ..., in)
```

SECTION 9

ARITHMETIC OPERATIONS

Calls to the routines which perform arithmetic are generated by the FORTRAN compiler when arithmetic operations are specified in the FORTRAN program. They should not be called explicitly by a FORTRAN program, but may be called in a PMA program.

All of these subroutines are callable in 32R or 64R mode and are contained in FTNLIB. The subset of these subroutines which are necessary in the 64V mode are in PFTNLB.

Subroutine names are of the form p$xy or F$pxy.

p is a prefix; x is the first argument (argument-1); y is the second argument (argument-2).

The prefix specifies the action of the subroutine (see Table 9-1). argument-1 is a number specifying the register in which the first argument is stored (see Table 9-2). argument-2 is a number specifying the type of the second argument pointed to by a DAC (R mode) or AP (V mode) following the subroutine call (see Table 9-2).

Note

In subroutines with only one argument, argument-2 has a slightly different meaning. This is discussed under the specific subroutines.

Examples: A$22 Adds two single-precision floating-point numbers (2 arguments).

C$12 Floats a 16-bit integer to a single-precision floating point number (1 argument).

A complete list of subroutines of this type follows:

A$21	C$26	D$51	E$27	F$DI11	F$SI11	M$77
A$51	C$27	D$52	E$51	F$DI71	F$SI71	
A$52	C$51	D$55	E$52	F$DI77	F$SI77	N$55
A$55	C$52	D$57	E$55			N$77
A$61	C$57	D$61	E$57	F$MA11	H$55	
A$62	C$61	D$62	E$61	F$MA22		S$21
A$77	C$62	D$67	E$62	F$MA77	L$55	S$51
	C$67	D$71	E$66			S$52
C$12	C$75	D$77	E$67	F$MI11	M$21	S$55
C$15	C$76		E$71	F$MI22	M$51	S$61
C$16	C$77	E$11	E$77	F$MI77	M$52	S$62
C$21		E$21			M$55	S$77
C$21G	D$21	E$22	F$CL	F$MO71	M$61	
C$25	D$27	E$26		F$MO77	M$62	Z$80

Table 9-1. Subroutine Prefix Explanations.

Prefix	Meaning	Number of Arguments
A	Addition	2
C	Conversion	1
D	Division	2
E	Exponentiation	2
H	Store complex number	1
L	Load complex number	1
M	Multiplication	2
N	Negation	1
S	Subtraction	2
Z	Zero double-precision exponent	1
	FORTRAN Support Subroutines (F$)	
DI	Positive difference	2
MA	Maximum	2
MI	Minimum	2
MO	Remainder	2
SI	Manitude of first times sign of second	2

Table 9-2. Data Type Codes.

Type code	Register	Type
1	A	16-bit integer (INTEGER*2)
2	FAC	Single-precision floating-point number (REAL or REAL*4)
5	AC1-AC4	Complex number (COMPLEX)
6	DFAC	double-precision floating-point number (DOUBLE PRECISION or REAL*8)
7	A+B	Long integer (INTEGER*4)
8	--	Exponent part of a Double-precision number

Note

Some long integer subroutines may need to be entered or exit in DBL mode (R mode only); this is noted with the description of these subroutines.

A	A register
FAC	Floating-point accumulator
AC1-AC4	Complex accumulator addresses AC1 to AC4
DFAC	Double-precision floating-point accumulator
A+B	Concontenated A and B registers

SINGLE ARGUMENT FUNCTIONS

Each of these subroutines takes a single argument, stored in the appropriate register, operates on it and stores the result in the same or another register.

Conversion

▶ C$xy

Converts the type of the argument in the register identified by x to the type of the argument identified by y and stores it in the proper register for y-type variables. For example C$75 converts a long integer in the A+B register into the real part of a complex number in the complex accumulator (imaginary part is Ø). See Table 9-3 for a complete list.

Complex Number Manipulation

▶ H$55

Stores the contents of the complex accumulator (AC1 to AC4) at the address specified by the DAC or AP following the call.

▶ L$55

Loads the complex accumulator (AC1 to AC4) from the four words pointed to by the DAC or AP following the call.

Negation

▶ N$xx

Negates the value of the argument in the register specified by x, and stores it in that same register (see Table 9-3).

Zeroing

▶ Z$8Ø

Clears the exponent part of the double-precision floating-point accumulator (DFAC). R mode only.

TWO-ARGUMENT SUBROUTINES

These subroutines perform arithmetic operations on two arguments: addition, subtraction, etc. If the arguments do not have the same data type, the data type of the result is that of the higher. The data types, in descending order are:

Table 9-3
Single Argument Subroutines
(Negation and Conversion)

x	y	N$ (Negation)	C$ (Conversion)
1	1		n/a
1	2	n/a	R
1	5	n/a	R,V
1	6	n/a	R
2	1	n/a	R
2	2		n/a
2	5	n/a	R,V
2	6	n/a	R
2	7	n/a	R
5	1	n/a	R,V
5	2	n/a	R,V
5	5	R,V	n/a
5	7	n/a	R,V
6	1	n/a	R
6	2	n/a	R
6	6		n/a
6	7	n/a	R,V
7	2	n/a	
7	5	n/a	R
7	6	n/a	R,V
7	7	R (1)	R

n/a Not applicable

R Used in R-mode only
R,V Used in R or V modes

x Argument type (see Table 9-2)
y Result type (see Table 9-2)

Notes

1. Exit mode is DBL (R mode).

2. There is also a subroutine C$21G (R mode only), which performs the same functions as C$21 without the use of any floating-point instructions.

COMPLEX or DOUBLE PRECISION
REAL
LONG INTEGER (INTEGER*4)
16-BIT INTEGER (INTEGER*2)

There are no operations which combine COMPLEX and DOUBLE PRECISION numbers, i.e., there are no "56" or "65" subroutines. The result of a two-argument subroutine is stored in the appropriate register for its data type (see Table 9-2).

Examples: R mode

```
CALL A$21
DAC I
```

Floats the 16-bit integer I and adds it to the contents of the Floating Point Accumulator (FAC).

V mode

```
CALL F$MIll
AP I2,SL
```

Loads I2 into the A register if I2 is less than the current contents of the A register.

Addition

▶ A$xy

Adds argument of type y, pointed to by the DAC or AP following the call, to an argument of type x in the appropriate register. See Table 9-4 for a complete list.

Division

▶ D$xy

Divides the argument of type x in the appropriate register by the argument of type y, pointed to by the DAC or AP following the call. See Table 9-4 for a complete list.

Exponentiation

▶ E$xy

Raises the argument of type x in the appropriate register to the power specified by the argument of type y pointed to by the DAC or AP following the call. A complete list is given in Table 9-4.

Note

In all modes zero to the zeroth power is one.

Multiplication

▶ M$xy

Multiplies the argument of type x in the appropriate register by the argument of type y pointed to by the DAC or AP following the call. See Table 9-4 for a complete list.

Subtraction

▶ S$xy

Subtracts the argument of type y, pointed to by a DAC or AP following the call, from an argument of type x in the appropriate register. See Table 9-4 for a complete list.

Positive Difference

▶ F$DIxy

Subtracts the argument of type y, pointed to by the DAC or AP following the call, from the argument of type x in the appropriate register. If the result is less than zero, the register is cleared. See Table 9-5 for a complete list.

Maximum

▶ F$MAxx

Places the maximum of the register specified by type x and the value of the argument of type x pointed to by the DAC or AP, into the specified register. See Table 9-5 for a complete list.

Minimum

▶ F$MIxx

Places the minimum of the register specified by type x and the value of the argument of type x pointed to by the DAC or AP, into the specified register. See Table 9-5 for a complete list.

Remainder

▶ F$MOxy

Divides an argument of type x in the appropriate register by an argument of type y, pointed to by the DAC or AP. The remainder is placed in the appropriate register. See Table 9-5 for a complete list.

Table 9-4
Two-Argument
Arithmetic Subroutines (First Group)

x	y	A$ Addition	S$ Subtraction	M$ Multiplication	D$ Division	E$ Exponentiation
1	1					R,V
2	1	R	R	R	R,V	R,V
2	2					R,V
2	6					R,V
2	7				R,V	R,V
5	1	R,V	R,V	R,V	R,V	R,V
5	2	R,V	R,V	R,V	R,V	R,V
5	5	R,V	R,V	R,V	R,V	R,V
5	7				R,V	R,V
6	1	R	R	R	R,V	R,V
6	2	R	R	R	R,V	R,V
6	6					R,V
6	7				R,V	R,V
7	1				R,V	R,V
7	7	R(1)	R(1)	R(1)	R(1)	R,V(1)

R Used in R mode only
R,V Used in R or V modes

x First argument, stored in appropriate register
y Second argument, pointed to by DAC (R mode) or AP (V mode)

Notes

1. Exit mode is DBL (R mode)

Table 9-5
Two-Argument
Arithmetic Subroutines (Second Group)

x	y	F$MO Remainder	F$SI Sign and Magnitude	F$DI Positive Difference	F$MA Maximum	F$MI Minimum
1	1		R,V	R,V	R,V	R,V
2	2				R,V	R,V
7	1	R,V	R,V	R,V		
7	7	R,V	R,V	R,V	R,V	R,V

R Used in R mode only
R,V Used in R or V modes

x First argument, stored in appropriate register
y Second argument, pointed to by DAC (R mode) or AP (V mode).

Sign and Magnitude

▶ F$SIxy

Multiplies the argument of type x in the appropriate register by the sign of the argument of type y pointed to by the DAC or AP and stores the result in the register for type x. See Table 9-5 for a complete list.

Comparison (R mode only)

▶ F$CL

Compares the long integer, L1, in the concatenated A and B registers with the long integer, L2, pointed to by a DAC following the call. Control passes as follows:

L1>L2	Next location
L1=L2	Skip one location
L1<L2	Skip two locations

The A and B registers are not modified.

```
Example:   CALL F$CL
           DAC L2
           ...return here if L1>L2
           ...return here if L1=L2
           ...return here if L1<L2
```

SECTION 10

MATHLB (FORTRAN MATRIX SUBROUTINES)

SCOPE OF MATHLB

MATHLB provides a set of subroutines that perform matrix operations, solve systems of simultaneous linear equations, and generate permutations and combinations of elements. See Table 10-1 for a summary.

SUBROUTINE CONVENTIONS

The following conventions are used in the subroutine descriptions in this section:

Names

All calls are shown with their double-precision, integer, and complex counterparts, if applicable, in brackets following the single-precision name. For example, if the single-precision name is XXXX, the double-precision, integer and complex names respectively are: DXXXX, IXXXX, and CXXXX.

Parameters

All parameters must be specified. Variables and arrays are assumed to be of the same mode as the subroutine (i.e., REAL, DOUBLE-PRECISION,INTEGER,COMPLEX). Matrix sizes and error flags must be declared INTEGER. Parameters enclosed in parentheses follow the names.

Arrays

Arrays are expected by MATHLB subroutines to be doubly subscripted arrays. The dimensions passed as arguments must agree with the array sizes declared in the calling program, or the elements cannot be properly accessed. Except where otherwise noted, when more than a single array is passed as an argument, the arrays may be the same array as in the calling program. For example, in matrix addition, it is permissible to specify: A = A + A

Work Arrays

Work arrays must always be distinct arrays in the calling program.

Table 10-1. Summary of Available Matrix Operations

Operation	Integer	Single Precision	Complex	Double Precision
Setting matrix to identify matrix	IMIDN	MIDN	CMIDN	DMIDN
Setting matrix to constant matrix	IMCON	MCON	CMCON	DMCON
Multiplying matrix by a scalar	IMSCL	MSCL	CMSCL	DMSCL
Matrix Addition	IMADD	MADD	CMADD	DMADD
Matrix Subtraction	IMSUB	MSUB	OMSUB	DMSUB
Matrix multiplication	IMMLT	MMLT	CMMLT	DMMLT
Calculating transpose matrix *	IMTRN	MTRN	CMTRN	DMTRN
Calculating adjoint matrix *	IMADJ	MADJ	CMADJ	DMADJ
Calculating inverted matrix *		MINV	CMINV	DMINV
Calculating signed cofactor *	IMCOF	MCOF	CMCOF	DMCOF
Calculating determinant *	IMDET	MDET	CMDET	DMDET
Solving a system of linear equations		LINEQ	CLINEQ	DLINEQ
Generating permutations	PERM			
Generating combinations	COMB			

* For square matrices only

▶ COMB

COMB computes the next combination of nr out of n elements with a single interchange each time it is called. The first call to comb returns the combination 1, 2, 3,...,nr. This subroutine is self-initializing and proceeds through all n!/(nr!*(n-nr)!) combinations. At the last combination, it returns a value of last = 1 and resets itself. The comb subroutine may be re-initialized by the user by passing a restrt value of 1 along with new values for n and nr. (The restrt parameter is optional; if re-initialization is not desired either omit this parameter from the calling sequence or set it to a value of Ø).

CALL COMB (icomb, n, nr, iwl, iw2, iw3, last, restrt)

	Mode	Subscript(s)	Dimension(s)	Comments
icomb	Integer	1	nr	return
n	Integer			pass
nr	Integer			pass
iwl	Integer	1	n	work
iw2	Integer	1	n	work
iw3	Integer	1	n	work
last	Integer			return
restrt	Integer			pass (optional)

Note

The calling program should not attempt to modify icomb, iwl, iw2, or iw3. For further details see: "Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations", Gideon, Ehrlich, Journal of the ACM, 2Ø, Number 3 (July 1973) pp. 5ØØ-5113.

▶ LINEQ

LINEQ solves the set of n-linear equations in n unknowns represented by (cmat) (xvect) = (yvect) where cmat is the nxn square matrix of coefficients, yvect is the nxl column vector of unknowns in which the solution is stored.

Note

For complex and double-precision numbers, use CLINEQ and DLINEQ respectively.

CALL {CLINEQ / LINEQ / DLINEQ} (xvect, yvect, cmat, work, n, npl, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
xvect	*	1	n	returned
yvect	*	1	n	passed
cmat	*	2	n,n	passed
work	*	2	npl, npl	work
n	Integer			passed
npl	Integer			passed (=N+1)
ierr	Integer			returned

* all of the same mode which determine the subroutine used.

The user is required to provide as a work area, a nplxnpl matrix work (npl = n+1). The integer error flag ierr returns one of three possible values.

ierr	
0	Solution found satisfactorily
1	Coefficient matrix singular
2	npl ≠ n+1

If ierr ≠ 0 no modifications are made to xvect.

▶ MADD

MADD adds the nxm matrix mat2 to the nxm matrix mat1 and returns the sum in a nxm matrix mats. in component form: mats (i,j) = mat1 (i,j) + mat2 (i,j) as i goes from 1 to n and j goes from 1 to m.

Note

For integer, complex and double-precision numbers use IMADD, CMADD, and DMADD respectively.

CALL { DMADD / CMADD / IMADD / MADD } (mats, mat1, mat2, n, m)

	Mode	Subscript(s)	Dimension(s)	Comments
mats	*	2	n,m	returned
mat1	*	2	n,m	passed
mat2	*	2	n,m	passed
n	Integer			passed
m	Integer			passed

* all of the same mode which determines the subroutine used

▶ MADJ

This subroutine calculates the adjoint of the nxn matrix mati and stores it in the nxn matrix mato. Each element of the output matrix is the signed cofactor of the corresponding element of the input matrix.

Note

For integer, complex, or double-precision numbers use IMADJ, CMADJ, or DMADJ respectively

CALL { MADJ / IMADJ / CMADJ / DMADJ } (mato, mati, n, iw1, iw2, iw3, iw4, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	returned
mati	*	2	n,n	passed
n	Integer			passed
iwl	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
ierr	Integer			returned

* all of the same mode which determines the subroutine used.

The error flag, ierr, may have one of two values:

ierr	
Ø	Adjoint successfully constructed
1	n<2 - no adjoint may be constructed

Note

mato and mati must be distinct.

▶ MCOF

Calculates the signed cofactor of the element mat (i,j) of the nxn matrix mat and stores this value in COF. If i = Ø and j = Ø the determinant of mat is calculated.

Note

For integers, complex, or double-precision numbers use IMCOF, CMCOF, or DMCOF respectively.

CALL { IMCOF / CMCOF / MCOF / DMCOF } (cof, mat, n, iwl, iw2, iw3, iw4, i, j, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
cof	*			returned
mat	*	2	n,n	passed
n	Integer			passed
iwl	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
i	Integer			passed
j	Integer			passed
ierr	Integer			returned

* all of the same mode which determines the subroutine used.

The integer error flag ierr has two possible values:

ierr

Ø	Cofactor calculated successfully
1	No cofactor calculated for any of the following reasons:

1) n<2 - no cofactor possible
2) i = j = n = 0 - no determinant
3) i = Ø and j ≠ Ø or i ≠ Ø and j = Ø - subscript error
4) i>n and/or j>n - subscript error

▶ MCON

This subroutine sets every element of the NxM matrix MAT equal to a constant CON.

Note

For integer, complex, or double-precision numbers use IMCON, CMCON, or DMCON respectively.

CALL { IMCON / MCON / CMCON / DMCON } (mat, n, m, con)

	Mode	Subscript(s)	Dimension(s)	Comments
mat	*	2	n,m	returned
n	Integer			passed
m	Integer			passed
con	*			passed

* all of the same mode which determines the subroutine used.

▶ MDET

Calculates the determinant of the nxn matrix mat and stores it in det.

Note

For integer, complex, or double-precision numbers use IMDET, CMDET, or DMDET respectively.

CALL { IMDET / MDET / CMDET / DMDET } (det, mat, n, iw1, iw2, iw3, iw4, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
det	*			returned
mat	*	2	n,n	passed
n	Integer			passed
iw1	*	1	n	work
iw2	*	1	n	work
iw3	*	1	n	work
iw4	*	1	n	work
ierr	Integer			returned

* all of the same mode which determines the subroutine used.

The integer error flag ierr may have one of two values:

ierr	
Ø	Determinant formed successfully
1	n = Ø - no determinant possible

▶ MIDN

This subroutine sets the nxn matrix mat equal to the nxn identity matrix. That is,

$$\begin{aligned} MAT\ (I,J) &= Ø,\ I \neq J \\ &= 1,\ I = J \end{aligned}$$

Note

For integer, complex, or double-precision numbers use IMIDN, CMIDN, or DMIDN respectively.

CALL { IMIDN / MIDN / CMIDN / DMIDN } (mat, n)

	Mode	Subscript(s)	Dimension(s)	Comments
mat	*	2	n,n	returned
n	Integer			passed

* the mode of this argument determines which subroutine is used and the representation of 1 in matrix.

mode	subroutine	representation of 1
integer	IMIDN	1
single-precision	MIDN	1.(SP)
complex	CMIDN	(1.,Ø) (each SP)
double-precision	DMIDN	1. (DP)

▶ MINV

Calculates the inverse of the nxn matrix mati and stores it in mato if successful. (The inverse of mati is mato if and only if

$$mati*mato = mato*mati = I$$

where * denotes matrix multiplication and I is the nxn identity matrix). The user must supply a npl x npn scratch matrix work, where npl = n+1 and npn = n+n.

Note

For complex or double-precision numbers use the subroutines CMINV or DMINV respectively. There is no integer form of this subroutine as there is no guarantee that the inverse of an integer matrix will be an integer matrix.

CALL {CMINV / MINV / DMINV} (mato, mati, n, work, npl, npn, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	returned
mati	*	2	n,n	passed
n	Integer			passed
work	*	2	npl,npn	work
npl	Integer			passed
npn	Integer			passed
ierr	Integer			returned

* all of the same mode which determines which subroutine is used.

The integer error flag ierr will return one of the following values.

ierr

Ø	matrix inverted - inverted matrix stored in mato.
1	matrix is singular - no inversion possible, mato is filled with zeroes.
2	npl ≠ n+1 and/or npn ≠ n+n - return from subroutines with no calculations performed.

▶ MMLT

This subroutine multiplies the n1xn2 matrix matl (on the left) by the n2xn3 matrix matr (on the right) and stores the resulting n1xn3 product matrix in matp.

Note

For integers, complex, or double-precision numbers use IMMLT, CMMLT, or DMMLT respectively.

CALL { IMMLT / MMLT / CMMLT / DMMLT } (matp, matl, matr, n1, n2, n3)

Note

matp must be distinct from matl and matr, although matl and matr may be the same. For example:

```
CALL MMLT (A, B, C, N1, N2, N3)     LEGAL
CALL MMLT (A, B, B, N, N, N)        LEGAL
CALL MMLT (A, A, A, N, N, N)        ILLEGAL
CALL MMLT (A, A, B, N, N, N)        ILLEGAL
CALL MMLT (A, B, A, N, N, N)        ILLEGAL
```

	Mode	Subscript(s)	Dimension(s)	Comments
matp	*	2	n1,n3	returned
matl	*	2	n1,n2	passed
matr	*	2	n2,n3	passed
n1	Integer			passed
n2	Integer			passed
n3	Integer			passed

* all of the same mode which determines which subroutine is used.

▶ MSCL

This subroutine multiplies the nxm matrix mati by the scalar constant SCON and stores the resulting nxm matrix in mato. By components scalar multiplication is understood to be: mato (i,j) = scon*mati (i,j) for i from 1 to n, j from 1 to m.

Note

For integers, complex, or double-precision numbers use IMSCL, CMSCL, or DMSCL.

CALL { IMSCL / MSCL / CMSCL / DMSCL } (mato, mati, n, m, scon)

	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,m	returned
mati	*	2	n,m	passed
n	Integer			passed
m	Integer			passed
scon	*			passed

* all of same mode which determines which subroutine is used.

▶ MSUB

Subtracts the nxm matrix mat2 from the nxm matrix mat1 and stores the difference in the nxm matrix matd.

Note

For integers, complex, or double-precision numbers use IMSUB, CMSUB, or DMSUB respectively.

CALL {IMSUB / MSUB / CMSUB / DMSUB} (matd, mat1, mat2, n, m)

	Mode	Subscript(s)	Dimension(s)	Comments
matd	*	2	n,m	returned
mat1	*	2	n,m	passed
mat2	*	2	n,m	passed
n	Integer			passed
m	Integer			passed

* all of the same mode which determines the subroutine to be used.

▶ MTRN

Calculates the transpose of the nxn matrix mati and stores it in the nxn matrix mato. The relationship between mati and mato is: mato (i,j) = mati (j,i) for i, j = 1 to n. mato and mati must be distinct.

Note

For integers, complex, or double-precision numbers use IMTRN, CMTRNM, or DMTRN respectively.

CALL { IMTRN / MTRN / CMTRN / DMTRN } (mato, mati, n)

	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	returned
mati	*	2	n,n	passed
n	Integer			passed

* all of the same mode which determines the subroutine used.

▶ PERM

PERM computes the next permutation of n elements with a single interchange of adjacent elements each time it is called. The first call to PERM returns the permutation 1, 2, 3,..., n. This subroutine is self-initializing and proceeds through all n! permutations. At the last permutation it returns a value of last = 1 and resets itself. The PERM subroutine may be re-initialized by the user by passing a new value of n or by passing the restrt parameter with a value of 1. (The restrt parameter is optional, if re-initialization is not desired either omit this parameter from the calling sequence or set it to a value of Ø.) The calling program should not attempt to modify iperm, iw1, iw2, or iw3.

CALL PERM (iperm, n, iw1, iw2, iw3, last, restrt)

	Mode	Subscript(s)	Dimension(s)	Comments
iperm	Integer	1	n	returned
n	Integer			pass
iw1	Integer	1	n	work
iw2	Integer	1	n	work
iw3	Integer	1	n	work
last	Integer			return
restrt	Integer			passed (optional)

For further details see:

"Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," Gideon, Ehrlich, Journal of the ACM, 20, Number 3 (July 1973) pp. 5000-5113.

SECTION 11

APPLICATION LIBRARY (APPLIB)

GENERAL DESCRIPTION

APPLIB is a user-oriented library which provides users with a set of service routines, designed for their ease-of-use. In many cases, the APPLIB routines call a lower-level routine filling in arguments that the caller isn't concerned about. The routines may also reformat the data that the lower-level routine returns. The use of APPLIB routines avoids a duplication of effort and provides a consistent interface for the terminal user.

All APPLIB routines are written as FORTRAN functions which return one of the following: a status indication (.TRUE. or .FALSE.), an appropriate value or an alternate value or format of a returned argument. The caller is never returned a "CODE" type argument which must then be decoded. All error detection, reporting, and, if possible, recovery are performed by the routine which returns only an indication of success or failure. This simplified error reporting scheme assures the user that the error is reported and all possible recovery procedures have been tried. In most cases, the reason for failure is an "irrelevant difference" and is ignored.

APPLIB ROUTINES

The categories and functions provided by the Application Library are:

String Manipulation Routines
User Query Routines
System Information Routines
Mathematical Routines
Conversion Routines
File System Routines
Parsing Routines

String Manipulation Routines

Compare two strings for equality.	CSTR$A
Compare two substrings for equality.	CSUB$A
Fill a string with a character (e.g. fill a name buffer with spaces).	FILL$A
Fill a substring with a given character.	FSUB$A
Get a character from a packed string.	GCHR$A
Left justify, right justify or center a string within a field.	JSTR$A
Locate one string within another.	LSTR$A

Locate one substring within another.	LSUB$A
Move a character from one packed string to another.	MCHR$A
Move one string to another.	MSTR$A
Move one substring to another.	MSUB$A
Determine the operational length of a string.	NLEN$A
Rotate string left or right.	RSTR$A
Rotate substring left or right.	RSUB$A
Shift string left or right.	SSTR$A
Shift substring left or right.	SSUB$A
Test for treename.	TREE$A
Determine string type.	TYPE$A

User Query Routines

Prompt and read a name.	RNAM$A
Prompt and read a number (binary, decimal, octal, or hexadecimal) into an INTEGER*4 variable.	RNUM$A
Ask question and obtain a yes or no answer.	YSNO$A

System Information Routines

CPU time since login.	CTIM$A
Today's date, American style.	DATE$A
Today's date as day of year ("Julian" date).	DOFY$A
Disk time since login.	DTIM$A
Today's date, European (Military) style.	EDAT$A
Time of day.	TIME$A

Mathematical Routines

Generate random number and update "seed". This generator is based upon a 32-bit word size and uses the Linear Congruential Method.	RAND$A
Initialize random number generator "seed".	RNDI$A

Conversion Routines

Convert a string from lower case to upper case or upper to lower.	CASE$A
Convert ASCII number to binary.	CNVA$A
Convent binary number to ASCII.	CNVB$A
ENCODE function that adjusts the "FORMAT" to make the number printable if possible. If not, the field is filled with asterisks.	ENCD$A

Convert the DATMOD field (as returned by RDEN$$)	FDAT$A
Convert the DATMOD field (as returned by RDEN$$) in different format than FDAT$A	FEDT$A
Convert the TIMMOD field (as returned by RDEN$$).	FTIM$A

File System Routines

Function	Subroutine
Close a file.	CLOS$A
Delete a file.	DELE$A
Check for file existence.	EXST$A
Position to end-of-file.	GEND$A
Open supplied name.	OPEN$A
Read name and open.	OPNP$A
Open supplied name with verification and delay.	OPNV$A
Read name and open with verification and delay.	OPVP$A
Position file.	POSN$A
Return position of file.	RPOS$A
Rewind file.	RWND$A
Open a scratch file with unique name.	TEMP$A
Truncate file.	TRNC$A
Scan the file system tree structure.	TSCN$A
Check for file open.	UNIT$A

Parsing Routine

Parse PRIMOS type command line.	CMDL$A

NAMING CONVENTIONS

All APPLIB routines follow a consistent naming convention designed to avoid the possibility of a conflict with user written routines and system routines. All APPLIB routines have a four letter mnemonic name and the suffix "$A". For example, the routine to open a temporary file is named "TEMP$A". Many routines have options which are specified by named "parameter" keys which all begin with the prefix "A$".

Subroutines that are used internally by APPLIB routines have a suffix of "$$A". These should not be used under ordinary circumstances.

All "parameter" keys are defined in a $INSERT file named SYSCOM>A$KEYS. The key names, following the "A$" prefix are three or four letter mnemonics specifying the allowable options for the various routines. In addition, this file supplies all the appropriate FUNCTION type declarations for the APPLIB routines. A complete listing of SYSCOM>A$KEYS is included at the end of this section.

APPLIB and its V-mode version, VAPPLB, exist as independent libraries in UFD=LIB.

LIBRARY IMPLEMENTATION AND POLICIES

The routines have been coded to make them easily callable from most other languages, including PL/I and 1977 ANSI FORTRAN, both of which can automatically generate string length arguments following string arguments. As a result, in the argument pair name, namlen, the name is often updated by an APPLIB routine, but the namlen argument is never modified. If the namlen argument is not zero or greater, an error message is displayed on the user terminal. Where applicable, the function value returned is .FALSE.. The function NLEN$A can be used to determine the operational length of a returned name.

All APPLIB routines which either accept keys as arguments or call other APPLIB routines which do, use the SYSCOM>A$KEYS file to define those keys. Also, these routines do not take advantage of any particular numerical values these keys may have, should it become necessary either to change these values or to add new keys with numerical values which do not fit the previous pattern. For example, there are no computed GOTO's on keys and no range checks for validity of a key. In this way, if a new SYSCOM>A$KEYS file is created, both the user programs and the routines they call will always agree on the meaning of a given key. The same is true of the declared types of the APPLIB functions.

Library Building

All routines are compiled into a single binary file which is then converted into the appropriate library file with the EDB utility. At present, the only difference between the R-mode and V-mode build procedures is the FTN compile option used. For APPLIB, all routines are compiled for 64R-mode loading. For VAPPLB, all routines are compiled for 64V-mode loading (SEG). In addition, all routines included in VAPPLB are pure procedure and may be loaded into the shared portion of a shared procedure.

Since several of the APPLIB routines call other APPLIB routines, the load order is important. This order is specified in the command files "C_APPLIB" and "C_VAPPLB" located in UFD = APPLIB.

STRING MANIPULATION ROUTINES - DETAILED DESCRIPTION

The string manipulation routines are designed to facilitate the handling of character strings. All of these routines operate on packed strings, unless stated otherwise. Most of the routines in this section require that the physical length of a string (in characters) be passed as an argument. The physical length is the actual storage allocated for that string in bytes or characters (including any trailing blanks). The operational length of a string does not include trailing blanks. Since the length of a string is specified as an INTEGER*2 variable, the maximum string length is 32,767 characters.

The majority of routines that operate on entire strings first truncate them to their operational length. The routines that operate on substrings treat any trailing blanks as part of the substring.

All string length specifications and substring delimiting character positions are checked for validity and must conform to the following rules.

1. Physical string length specifications must be greater than or equal to zero. A value of zero indicates a null or empty string.

2. Substring delimiting character positions must be greater than or equal to zero. The length of the substring must be less than or equal to the physical string length. The beginning character position must be less than or equal to the ending character position. A value of zero for either the starting or ending character position indicates a null substring.

If these rules are violated, an error message will be displayed and the logical functions will be .FALSE..

▶ CSTR$A

CSTR$A is a logical function used to compare two strings for equality. The function will be .TRUE. if each character in string a matches the corresponding character in string b, or if both strings are null (i.e., length equal to zero). Otherwise, the function will be .FALSE.. Only the operational lengths are used in the comparison (i.e., trailing blanks are ignored).

LOG = CSTR$A(a,alen,b,blen)

a String to be compared, packed two characters per word.

alen Length of a, in characters (INTEGER*2). Length must be zero or greater.

b String to be compared against, packed two characters per word.

blen Length of b, in characters (INTEGER*2). Length must be zero or greater.

APPLIB CALLS: CSUB$A, NLEN$A

▶ CSUB$A

CSUB$A is a logical function used to compare substrings for equality.

LOG = CSUB$A(a,alen,afc,alc,b,blen,bfc,blc)

a	Array containing substring to be compared, packed two characters per word. Data type does not matter.
alen	Length of a, in characters (INTEGER*2). Length must be zero or greater.
afc	First character position of substring in a (INTEGER*2).
alc	Last character position of substring in a (INTEGER*2).
b	Array containing substring to be compared against, packed two characters per word. Data type does not matter.
blen	Length of b, in characters (INTEGER*2), must be zero or greater.
bfc	First character position of substring in b (INTEGER*2).
blc	Last character position of substring in b (INTEGER*2).

If each character in the a substring matches the corresponding character in the b substring, or both substrings are null (i.e., length equal to zero) the function will be .TRUE.. If two corresponding characters do not match, or if the lengths of the substrings are not equal the function will be .FALSE..

APPLIB CALLS: None

▶ FILL$A

FILL$A is an INTEGER function which fills the name buffer with the fill character supplied. The function is INTEGER, but value is always Ø.

```
INT =FILL$A(name,namlen,char)
CALL FILL$A(name,namlen,char)
```

name	Name of buffer to fill packed two characters per word. Data type does not matter.
namlen	Length of name in characters (INTEGER*2).
char	Fill character in FORTRAN A1 format. Data type does not matter.

APPLIB CALLS: None

▶ FSUB$A

FSUB$A is a logical function used to fill a character substring with a specified character. The substring delimited by fchar and lchar are filled with the character specified in filchar. The string parameters are checked for validity and if an error is found, the function will be .FALSE. and a message is printed. If all parameters are valid, the function will be .TRUE..

```
LOG = FSUB$A(string, length, fchar, lchar, filchar)
CALL  FSUB$A(string, length, fchar, lchar, filchar)
```

string	string containing substring to be filled, packed two characters per word. Data type does not matter.
length	length of string in characters (INTEGER*2).
fchar	first character position of substring (INTEGER*2).
lchar	last character position of substring (INTEGER*2).
filchar	fill character in FORTRAN A1 format. Data type does not matter.

APPLIB CALLS: None

▶ GCHR$A

GCHR$A is an INTEGER function which extracts a single character from a packed string. The function value will be the accessed character in FORTRAN A1 format. The character returned will be left-justified and padded with blanks.

```
INT GCHR$A(farray,fchar)
CALL GCHR$A(farray,fchar)
```

farray	Source packed array. Data type does not matter.
fchar	Character position in farray to be returned (INTEGER*2).

This routine replaces the FORTRAN statement:

```
CHAR = FARRAY(FCHAR)
```

When FARRAY is declared LOGICAL*1 (IBM FORTRAN) or of a one character data type.

APPLIB CALLS: None

▶ JSTR$A

JSTR$A is a logical function used to left justify, right justify or center a string within itself.

```
LOG = JSTR$A(key,string,length)
CALL  JSTR$A(key,string,length)
```

key — Determines direction of justification, possible values are:

A$RGHT - right justify.

A$LEFT - left justify.

A$CNTR - center.

string — String to be justified, packed two characters per word. Data type does not matter.

length — Length of string in characters (INTEGER*2), must be .GE. zero.

The function will be .TRUE. if justification is successful, .FALSE. if the string length is less than zero or if a bad key is used.

APPLIB CALLS: NLEN$A, FILL$A, MSUB$A, GCHR$A

▶ LSTR$A

LSTR$A is a logical function used to locate one string within another.

```
LOG = LSTR$A(a,alen,b,blen,fcp,lcp)
CALL  LSTR$A(a,alen,b,blen,fcp,lcp)
```

a — String to be located, packed two characters per word. Data type does not matter.

alen — Length of a, in character (INTEGER*2).

b — String to be searched, packed two characters per word. Data type does not matter.

blen — Length of b, in characters (INTEGER*2).

fcp — First character position in b of substring that matches string a (INTEGER*2).

lcp — Last character position in b of substring that matches string a (INTEGER*2).

LSTR$A will search string b for the first occurrence of string a. If string a is found, the function will be .TRUE. and fcp and lcp will be equal to the character positions of the substring in b that matches string a. If string a is not found or if either string is null (i.e., length equal to zero) the function will be .FALSE. and fcp and lcp will be equal to zero. Each string is logically truncated to its operational length before the search is performed (i.e., trailing blanks are ignored).

APPLIB CALLS: LSUB$A, NLEN$A

▶ LSUB$A

LSUB$A is a logical function used to locate one substring within another.

```
LOG = LSUB$A(a,alen,afc,alc,b,blen,bfc,blc,fcp,lcp)
CALL  LSUB$A(a,alen,afc,alc,b,blen,bfc,blc,fcp,lcp)
```

- a — Array containing substring to be located, packed two characters per word. Data type does not matter.
- alen — Length of a, in characters (INTEGER*2).
- afc — First character position of substring in a (INTEGER*2).
- alc — Last character position of substring in a (INTEGER*2).
- b — Array containing substring to be searched packed two characters per word. Data type does not matter.
- blen — Length of b, in characters (INTEGER*2).
- bfc — First character position of substring in b (INTEGER*2).
- blc — Last character position of substring in b (INTEGER*2).
- fcp — First character position in b of substring that matches substring in a (INTEGER*2).
- lcp — Last character position in b of substring that matches substring in a (INTEGER*2).

LSUB$A searches the substring contained in b for the first occurrence of the substring contained in a. If a match is found, fcp and lcp will be equal to the character positions in b of the matching substring and the function is .TRUE.. If a matching substring cannot be found or if either substring is null (i.e. length equal to zero) the function will be .FALSE. and fcp and lcp will be equal to zero.

APPLIB CALLS: None

▶ MCHR$A

MCHR$A is an INTEGER function which moves a character from one packed string to another.

```
CALL MCHR$A(tarray,tchar,farray,fchar)
I*2= MCHR$A(tarray,tchar,farray,fchar)
I*4= MCHR$A(tarray,tchar,farray,fchar)
```

tarray	Receiving array of characters packed 2 per word, first character on the left. This constitutes an APPLIB standard string. (typeless)
tchar	Character position in tarray of received character. (INTEGER*2)
farray	Source string. Data type does not matter.
fchar	Source character position in farray. (INTEGER*2)

This routine replaces the FORTRAN statement:

```
TARRAY(TCHAR) = FARRAY(FCHAR)
```

when TARRAY and FARRAY are declared LOGICAL*1 (IBM FORTRAN) or of a one character data type. Only the TCHAR'th character in TARRAY is replaced.

The function value will be the character that was moved in FORTRAN A1 format, that is, the character in the left-most byte, right padded with blanks.

APPLIB CALLS: None

▶ MSTR$A

MSTR$A is an integer function used to move the source string to the destination string.

```
INT = MSTR$A(a,alen,b,blen)
CALL  MSTR$A(a,alen,b,blen)
```

a — Source string, packed two characters per word. Data type does not matter.

alen — Length of a, in characters (INTEGER*2).

b — Destination string, packed two characters per word. Data type does not matter.

blen — Length of b, in characters (INTEGER*2).

If the source string is longer than the destination string it will be truncated. If it is shorter, it will be padded with blanks. The source and destination strings may overlap. The function value will be equal to the number of characters moved (excluding blank padding). If either string is null (i.e., length equal to zero) no characters are moved and the function value will be equal to zero.

APPLIB CALLS: MSUB$A

▶ MSUB$A

MSUB$A is an integer function used to move the source substring contained in a to the destination substring contained in b.

```
INT = MSUB$A(a,alen,afc,alc,b,blen,bfc,blc)
CALL  MSUB$A(a,alen,afc,alc,b,blen,bfc,blc)
```

a — Array containing source substring, packed two characters per word. Data type does not matter.

alen — Length of a, in characters (INTEGER*2).

afc — First character position of substring in a packed two characters per word. Data type does not matter.

alc — Last character position of substring in a (INTEGER*2).

b — Array containing destination substring, packed two characters per word. Data type does not matter.

blen — Length of b, in characters (INTEGER*2).

bfc First character position of substring in b (INTEGER*2).

blc Last character position of substring in b (INTEGER*2).

If the source substring is longer than the destination substring it will be truncated. If it is shorter it will be padded with blanks. The source and destination substrings may overlap.

If either substring is null (ie. length equal to zero) no characters are moved and the function will be equal to zero. Otherwise it is equal to the number of characters moved (excluding blanks used for padding).

APPLIB CALLS: MCHR$A

▶ NLEN$A

NLEN$A is an INTEGER*2 function which returns, as its function value, the operational length (not including trailing blanks) of the name in name.

```
I*2= NLEN$A(name,namlen)
CALL NLEN$A(name,namlen)
```

name Name buffer to be tested, packed two characters per word. Data type does not matter.

namlen Length of name in characters. (INTEGER*2)

APPLIB CALLS: None

▶ RSTR$A

RSTR$A is a logical function used to rotate a character string left or right. The string is truncated to its operational length before the rotate is performed, therefore trailing blanks are not included in count. If length is less than zero, the function will be .FALSE., otherwise function will be .TRUE..

```
LOG = RSTR$A(string, length, count)
CALL  RSTR$A(string, length, count)
```

string String to be rotated, packed two characters per word. Data type does not matter.

length Length of string in characters (INTEGER*2).

count Number of positions to rotate string. Negative count causes left rotate, positive count right rotate (INTEGER*2).

This routine uses an algorithm that minimizes temporary storage and execution time. One word of temporary storage is used and the number of iterations necessary to rotate a string is equal to the length in characters of the string. A character is moved directly from its original position to its final destination position.

APPLIB CALLS: MCHR$A, NLEN$A

▶ RSUB$A

RSUB$A is a logical function used to rotate a character substring left or right. Only the characters of the substring, contained in string are affected. The parameters are checked for validity and if there is an error, a message is printed and the function will be .FALSE.. If no error occurs, the function will be .TRUE..

```
LOG = RSUB$A(string, length, fchar, lchar, count)
CALL  RSUB$A(string, length, fchar, lchar, count)
```

string	String containing substring to be rotated, packed two characters per word. Data type does not matter.
length	Length of string in characters (INTEGER*2).
fchar	First delimiting character position of substring (INTEGER*2).
lchar	Last delimiting character position of substring (INTEGER*2).
count	Number of positions to rotate substring. Negative count causes left rotate, positive count causes right rotate (INTEGER*2).

This routine uses an algorithm that minimizes temporary storage and execution time. One word of temporary storage is used and the number of iterations necessary to rotate a string is equal to the length in characters of the string. A character is moved directly from its original position to its final destination position.

APPLIB CALLS: MCHR$A

▶ SSTR$A

SSTR$A is a logical function used to shift a character string left or right. The string is shifted the specified number of characters and the vacated positions are padded with the specified fill character. Trailing blanks are not included in the shift. If length is less than zero, an error message will be printed and the function will be .FALSE., and no characters are shifted. If no error occurs, the function will be .TRUE..

```
LOG = SSTR$A(string, length, count, filchr)
CALL  SSTR$A(string, length, count, filchr)
```

string — Character string to be shifted, packed two characters per word. Data type does not matter.

length — Length of string in characters. Must be greater than or equal to zero (INTEGER*2).

count — Number of positions to shift string. Negative count causes left shift, positive count causes right shift (INTEGER*2).

filchr — Fill character which will pad the vacated positions. Filchr is specified in A1 format. Data type does not matter.

APPLIB CALLS: FSUB$A, MCHR$A, NLEN$A

▶ SSUB$A

SSUB$A is a logical function used to shift a character substring left or right. The substring is shifted the specified number of characters and the vacated positions are padded with the specified fill character. Any trailing blanks are included in the shift. The parameters are checked for validity and an error will cause a message to be printed and the function will be .FALSE.. If no error occurs, the function will be .TRUE.. If the substring is null, or length equal to zero, there will be no shift.

```
LOG = SSUB$A(string, length, fchar, lchar, count, filchar)
CALL  SSUB$A(string, length, fchar, lchar, count, filchar)
```

string — String containing substring to be shifted, packed two characters per word. Data type does not matter.

length — Length of string in characters (INTEGER*2).

fchar — First delimiting character position of substring (INTEGER*2).

lchar — Last delimiting character position of substring (INTEGER*2).

count — Number of positions to shift substring. Negative count causes left shift, positive count causes right shift (INTEGER*2).

filchar Fill character with which to pad the vacated positions. Filchar is specified in A1 format. Data type does not matter.

APPLIB CALLS: FSUB$A, MCHR$A

▶ TREE$A

TREE$A is a logical function which scans a file name and determines if it is a tree name. If it is a tree name, the function is .TRUE. and if not, it is .FALSE.. In addition, the final name (or entire name if not in a tree) is located in the string. Note that if the name is empty, FSTART=FLEN=0.

LOG = TREE$A(name,namlen,fstart,flen)

name Array containing file name, packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2).

fstart Character position in name of first character in final name. (INTEGER*2)

flen Length of final file name in characters (INTEGER*2).

APPLIB CALLS: GCHR$A, NLEN$A

▶ TYPE$A

TYPE$A is a logical function which will test a character string to determine if it can be interpreted as the type specified by key.

LOG = TYPE$A(key,string,length)

key String type to be tested for, possible keys are:

A$NAME can string be interpreted as a name,

A$BIN can string be interpreted as a binary number,

A$DEC can string be interpreted as a decimal number,

A$OCT can string be interpreted as an octal number,

A$HEX can string be interpreted as a hexadecimal number.

string String to be tested, packed two characters per word. Data type does not matter.

length Length of string, in characters (INTEGER*2).

A string is interpreted as a name if it contains at least one alphabetic or special character (other than a leading + or -) a binary number if it contains only the digits Ø - 9, a decimal number if it contains only the digits Ø - 9. It is an octal number if it contains only the digits Ø - 7, and is a hexadecimal number if it contains only the digits Ø - 9 and the characters A - F (upper case only). A number may have a leading sign and any number of blanks between the sign and the first digit. However imbedded blanks within the number itself are not allowed. A number must also have at least one digit.

Leading and trailing blanks are ignored. The function is .TRUE. if string satisfies the conditions required by the key used; otherwise it is .FALSE. A null string (i.e., length equal to zero) will only return a function value of true if key is A$NAME.

APPLIB CALLS: GCHR$A, NLEN$A

USER QUERY ROUTINES - DETAILED DESCRIPTION

The user query routines provide a convenient means to input data from the user's terminal. Each routine has the ability to prompt the terminal user with a supplied message and then process his response.

▶ RNAM$A

RNAM$A is a logical function which prints the supplied message prompt and appends the characters ":" to it. It then reads a user response from the command stream. If the response is not a legal name, or if the name provided is too long for the supplied buffer, an error message will be typed and the message prompt will be repeated. If no name is provided, the value of the function will be .FALSE.. If a legal name is provided, the function value will be .TRUE.. The caller should be aware that COMANL and RDTK$$ (Section 5) are called to read the user response, and therefore the previous command line entered is unavailable.

LOG = RNAM$A(msg,msglen,namkey,name,namlen)

msg Message text, packed two characters per word. Data type does not matter.

msglen Message length in characters (INTEGER*2).

namkey A$FUPP, force upper case.

A$UPLW, do not force upper case.

A$RAWI, read line as raw uninterpreted text (keys cannot be combined).

name Returned name, packed two characters per word. Data type does not matter.

namlen Length of name buffer in characters (.LE. 80) (INTEGER*2).

APPLIB CALLS: None

▶ RNUM$A

RNUM$A is a logical function used to input numeric data from the user terminal. The routine prints the user-supplied message and appends the character ":" to it. It then reads a user response and if the response is not a legal number or if the number provided has too many digits for an INTEGER*4 value, the error will be reported and the message will be repeated. If no number is provided, the value of the function will be .FALSE. and VALUE=0. If a legal number is provided, the function will be .TRUE. and the value will be returned in value. The caller should be aware that COMANL and RDTK$$ (Section 5) are called to read the user response, and therefore the previous command line is unavailable. Numbers may be immediately preceded by "+" or "-". Binary numbers may have a maximum of 31 digits, octal a maximum of 11 digits, decimal a maximum of 10 digits and hexadecimal a maximum of 8 digits. Negative binary octal, or hexadecimal should not be entered in 2's complement, but the same as a negative decimal number.

LOG = RNUM$A(msg,msglen,numkey,value)

msg Message text, packed two characters per word. Data type does not matter.

msglen Message length in characters (INTEGER*2).

numkey A$DEC, decimal.

A$BIN, binary.

A$OCT, octal.

A$HEX hexadecimal.

value Returned value.

APPLIB CALLS: None

▶ YSNO$A

YSNO$A is a logical function which prints the supplied message and appends the character "?" to it. It then reads a user response. If the answer is "YES" or "OK", the function value is .TRUE.. If the answer is "NO", the function value is .FALSE.. If an illegal answer is provided or if no default is accepted, the message will be repeated. User responses may be abbreviated to the first 1 or 2 characters.

LOG = YSNO$A(msg,msglen,defkey)

msg	Message text, packed two characters per word. Data type does not matter.
msglen	Message length in characters (INTEGER*2).
defkey	A$NDEF, no default accepted.
	A$DNO, default = "NO".
	A$DYES, default = "YES".

APPLIB CALLS: None

SYSTEM INFORMATION ROUTINES - DETAILED DESCRIPTION

The system information routines return the system date, system time, CPU time and disk time in character string format.

▶ CTIM$A

CTIM$A is a Double Precision function which returns CPU time elapsed since login, in seconds, and as centiseconds in the cputim argument.

```
R*8 = CTIM$A(cputim)
CALL  CTIM$A(cputim)
```

cputim CPU time in centiseconds (INTEGER*4).

The function value will be CPU time elapsed since login, in seconds. This value may be received as either REAL*4 or REAL*8.

APPLIB CALLS: None

▶ DATE$A

DATE$A is a Double Precision function which returns the date in the form "DAY, MON DD YEAR". The value of the function is the date in the form "MM/DD/YY". This value must be received as REAL*8.

Note that this routine is good for the period January 1, 1977, through December 31, 2076.

```
R*8 = DATE$A(date)
CALL  DATE$A(date)
```

date Date in the form "DAY, MON DD YEAR". Data type does not matter as long as it is at least 16 characters long.

APPLIB CALLS: None

▶ DOFY$A

DOFY$A is a Double Precision function which returns the day of the year in the form "DDD". The value of the function is the date in the form YR.DDD suitable for printing in FORMAT F6.3. This value can be received as either REAL*4 or REAL*8. This routine is good for the period January 1, 1977 through December 31, 2076.

```
R*8 = DOFY$A(dofy)
CALL  DOFY$A(dofy)
```

dofy Day of year in the form "DDD" ("Julian" date). The data type does not matter as long as it is at least 4 characters long.

APPLIB CALLS: None

▶ DTIM$A

DTIM$A is a Double Precision function which returns disk time since login as INTEGER*4 centiseconds in the dsktim argument. The function value will be disk time since login in seconds. This value may be received as either REAL*4 or REAL*8.

```
R*8 = DTIM$A(dsktim)
CALL  DTIM$A(dsktim)
```

dsktim Disk time in centiseconds (INTEGER*4).

APPLIB CALLS: None

▶ EDAT$A

EDAT$A is a Double Precision function which returns the date in the European (military) form DAY, 'DD MON YEAR' in edate. The value of the function is the date in the form 'DD/MM/YY'. This value must be received in a REAL*8 variable. The routine is good for the period 1 January 1977 through 31 December 2076.

```
R*8 = EDAT$A(edate)
CALL  EDAT$A(edate)
```

edate Date in the form "DAY, DD MON YEAR".

The type of the edate array does not matter as long as it is at least 16 characters long.

APPLIB CALLS: DATE$A

▶ TIME$A

TIME$A is a Double Precision function which returns the time of day in the form HR:MN:SC. The value of the function is the time of day in decimal hours. This value may be received as either REAL*4 or REAL*8.

```
R*8 = TIME$A(time)
CALL  TIME$A(time)
```

time Time of day in the form HR:MN:SC packed two characters per word. Data type does not matter as long as it is at least 8 characters long.

APPLIB CALLS: None

MATHEMATICAL ROUTINES - DETAILED DESCRIPTION

The mathematical routines provide miscellaneous functions not available in MATHLIB.

▶ RAND$A

RAND$A is a Double Precision function which updates a seed to a new seed (SEED) based upon the linear congruential method:

U(I)=FLOAT(K(I))/M

K(I) B*K(I-1) modulo M

B 16807.0

M 2**31-1 = 2147483647.0

B and M are from: Lewis, Goodman, and Miller, "A Pseudo-random Number Generator for the System/360", IBM Systems Journal, Vol. 8, No. 2, 1969, pp. 136-145.

K(I-1) is the input value of seed and K(I) is the returned value.

The value of the function is U(I) which represents a probability and is between 0.0 and 1.0. This value may be received as either REAL*4 or REAL*8.

```
R*4 = RAND$A(seed)
R*8 = RAND$A(seed)
CALL  RAND$A(seed)
```

seed Input is previous seed, output is new seed (INTEGER*4).

APPLIB CALLS: None

▶ RNDI$A

RNDI$A is a Double Precision function which returns the time of day in centiseconds. The function value will be the time of day in seconds. This value may be received as either REAL*4 or REAL*8.

Note: Because this function is used to initialize a random number generator, if the value is exactly 0, 1234567 and 12345.67 will be returned instead.

```
R*4 = RNDI$A(seed)
R*8 = RNDI$A(seed)
CALL  RNDI$A(seed)
```

seed Time of day in centiseconds (INTEGER*4).

APPLIB CALLS: None

CONVERSION ROUTINES - DETAILED DESCRIPTION

▶ CASE$A

CASE$A is a logical function which converts a string from upper case to lower, or from lower case to upper. The function will be .FALSE. if length is less than zero, otherwise .TRUE..

```
LOG = CASE$A(key, string, length)
CALL  CASE$A(key, string, length)
```

key — A$FUPP, convert all alphabetic characters in string from lower case to upper case.

A$FLOW, convert all alphabetic characters in string from upper case to lower case.

Default: No conversion.

string — Array containing character string to be converted, packed two characters per word. Data type does not matter.

length — Length of string in characters (INTEGER*2).

APPLIB CALLS: GCHR$A, MCHR$A

▶ CNVA$A

CNVA$A is a logical function that converts an ASCII digit string into its binary value for decimal, octal and hexadecimal numbers. The

numbers may be explicitly signed. Leading and trailing blanks are ignored as well as blanks between the sign and the number. However, blanks within the number are not allowed. If the number converts successfully, the function is .TRUE. and value is the converted binary value. If conversion is not successful, the function is .FALSE. and value=Ø. Note that for decimal conversions, overflow will be considered as unsuccessful whereas for octal and hexadecimal conversions, overflow is ignored.

```
LOG = CNVA$A(numkey,name,namlen,value)
CALL  CNVA$A(numkey,name,namlen,value)
```

numkey A$DEC, decimal
A$BIN, binary
A$OCT, octal
A$HEX, hexadecimal.

name Array containing ASCII digit string, packed two characters per word. Data type does not matter. Maximum digits are: binary, 31 - octal, 11 - decimal, 1Ø - hexadecimal, 8. Maximum does not include leading signs or blanks.

namlen Length of name in characters(INTEGER*2).

value Returned converted binary value (INTEGER*2).

APPLIB CALLS: GCHR$A, NLEN$A

▶ CNVB$A

CNVB$A is an integer function used to convert a binary number to an ASCII digit string.

```
I*2 = CNVB$A(numkey,value,name,namlen)
CALL  CNVB$A(numkey,value,name,namelen)
```

numkey Number base to convert to; possible values are:

A$BIN binary number with leading blanks.

A$BINZ binary number with leading zeros.

A$DEC signed decimal number with leading blanks.

A$DECU unsigned decimal number with leading blanks.

A$DECZ signed decimal number with leading zeros.

A$OCT octal number, leading blanks.

A$OCTZ octal. leading zeros,

A$HEX hexadecimal, leading blanks.

A$HEXZ hexadecimal, leading zeros.

name array containing returned ASCII digit string packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2). Maximum length for binary is 31, octal is 11, decimal is 10, and hexadecimal is 8. Maximum does not include leading signs or zeros.

value Binary number to be converted (INTEGER*4).

CNVB$A will convert a binary number into an ASCII digit string for decimal, octal, and hexadecimal numbers. The returned digit string will be right justified in name and preceded by leading blanks or zeros depending upon numkey specification.

If value is negative and the number is to be treated as signed decimal, the digit will begin with an initial "-" sign. If value is negative, binary, octal and hexadecimal numbers will be in 2's complement form. If the number converts successfully, the function value is the number of digits and if not, it is zero.

APPLIB CALLS: FILL$A, MCHR$A

▶ ENCD$A

ENCD$A is a logical function which attempts to encode value in the supplied Fw.d format if it will fit. If not, the dec argument is decremented (moving the decimal point to the right) until it will fit. If dec reaches 0, or is originally supplied as 0, value will be encoded in Iw format if the number will fit into a 32-bit integer. If not, and if the field is wide enough (width > 7), the value will be encoded in E format. If the field is not wide enough, it will be filled with asterisks.

Note that the largest value of width will be 16. If it is larger than 16, only the first 16 characters of array will be used.

The function value will be .TRUE. if the encode was successful, and .FALSE. if the field was filled with asterisks.

Note that array is the only argument which is actually modified in the calling program.

```
LOG = ENCD$A(array,width,dec,value)
CALL  ENCD$A(array,width,dec,value)
```

array Array to receive value, packed two characters per word. Data type does not matter.

width Field width as in format Fw.d (should be even) (INTEGER*2).

dec Places to right of decimal point as shown in format Fw.d (INTEGER*2).

value Double precision value to be encoded (REAL*8).

APPLIB CALLS: None

▶ FDAT$A

FDAT$A is a REAL*8 function which converts the datmod field, returned by RDEN$$, to the format 'DAY, MON DD YEAR'. The function value is the datmod field converted to 'MM/DD/YY' and must be received in a REAL*8 variable. The routine is good for the period 1 January, 1972 to 31 December, 2071.

```
CALL  FDAT$A(datmod,date)
R*8   FDAT$A(datmod,date)
```

datmod Date returned by RDEN$$. This is date the file was last modified and is in the format YYYYYYYMMMMDDDDD. YYYYYYY is the year modulo 100, MMMM is the month and DDDDD is the day (INTEGER*2).

date Array containing the date as a character string, packed two characters per word. Date is in format 'DAY, MON DD YEAR'. Data type does not matter as long as array is at least 16 characters long.

APPLIB CALLS: CNVB$A

▶ FEDT$A

FEDT$A converts the datmod field, returned by RDEN$$, to the format 'DAY, MON DD YEAR' in date. The function value is datmod converted to 'MM.DD.YY' and must be reserved in a REAL*8 variable. The routine includes the period 1 January, 1972 - 31 December, 2071.

```
CALL  FEDT$A(datmod,date)
R*8   FEDT$A(datmod,date)
```

datmod — Date returned by RDEN$$. This is date the file was last modified and is in the format YYYYYYYMMMMDDDDD. YYYYYYY is the year modulo 100, MMMM is the month and DDDDD is the day (INTEGER*2).

date — Array containing the date as a character string, packed two characters per word. Date is in the format 'DAY, MON DD YEAR'. Data type does not matter as long as array is at least 16 characters long.

APPLIB CALLS: FDAT$A

▶ FTIM$A

FTIM$A is a REAL*4 or REAL*8 function which converts the timmod field, returned by RDEN$$, to the format 'HH:MM:SS'. The function value is the timmod field converted to decimal hours and may be received as either REAL*4 or REAL*8.

```
CALL  FTIM$A(timmod,time)
R*8   FTIM$A(timmod,time)
R*4   FTIM$A(timmod,time)
```

timmod — Time at which a file was last modified, in the format 'seconds since midnight' divided by 4 (INTEGER*2).

time — Array containing time a file was last modified, as a character string in the format 'HH:MM:SS'. Data type does not matter as long as array is at least 8 characters long.

APPLIB CALLS: CNVB$A

FILE SYSTEM ROUTINES - DETAILED DESCRIPTION

The file system routines in APPLIB give the user a simple and consistent way to specify the most common file system operations. Accordingly, APPLIB does not provide the user with the full capabilities of the file system routines since for detailed operations it is best to use the file system routines, themselves. APPLIB supports both Sequential Access Method (SAM) and Direct Access Method (DAM) files. There is no support for segment directory type files as the MIDAS subsystem provides the higher level functions with these files.

All routines except Open, Delete, and Check for File Existence use only the file unit and not the file name. Also, each routine carries the name of its function, as above, with arguments consisting of only the relevant information, usually only the unit number. Note that all file names, except scratch files, may be pathnames.

The only complicated routines are the five Open routines, because of the many ways programs can obtain the name of the file they wish to open and the various options for verification or error recovery. Five different routines exist to perform the varying levels of complexity. In this way, the simple operations are represented by simple calling sequences. Only complex operations need complex argument lists.

All OPEN routines allow selection of the file type (SAM or DAM) and all but TEMP$A allow specification of the open mode (READ, WRITE, or READ/WRITE). TEMP$A (scratch) files are always opened for READ/WRITE.

All OPEN routines can choose the file unit number upon which a file will be opened. The A$GETU key is used for this purpose and the file unit selected by the routine will be returned to the user (in the argument unit). If A$GETU is not used, the user must provide the routine with a usable file unit number.

Verification provides the following options:

1. Verifies that the file is new; otherwise, verifies that it is O.K. to modify a file which already exists.

2. Verifies that file may be modified and determines whether an existing file is to be overwritten or appended.

3. Verifies that the file is old; that is, does not allow creation of a new file. Note that if the open mode is READ, this is the only possible verification option.

Delay provides the following options:

1. If and only if the file is "IN USE", waits a supplied number of seconds (elapsed time) and tries again.

2. The ability to retry 1 above a specified number of times.

▶ CLOS$A

CLOS$A is a logical function that closes the file open on unit. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = CLOS$A(unit)
CALL  CLOS$A(unit)
```

unit File unit. (INTEGER*2)

APPLIB CALLS: None

▶ DELE$A

DELE$A is a logical function which deletes the file named in name. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = DELE$A(name,namlen)
CALL  DELE$A(name,namlen)
```

name File name (may be a treename) packed two characters per word. Data type does not matter.

namlen Length of name in characters. (Mode is INTEGER*2.)

APPLIB CALLS: TREE$A, UNIT$A, NLEN$A

▶ EXST$A

EXST$A is a logical function which returns .TRUE. if the file exists and .FALSE. if the file does not exist or if an error was encountered.

```
LOG = EXST$A(name,namlen)
```

name File name (may be a treename) packed two characters per word. Data type does not matter.

namlen Length of name in characters. (Mode is INTEGER*2.)

APPLIB CALLS: TREE$A, UNIT$A, NLEN$A

▶ GEND$A

GEND$A is a logical function which positions the file open on unit to End-of-File. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = GEND$A(unit)
CALL  GEND$A(unit)
```

unit PRIMOS file unit. (Mode is INTEGER*2.)

APPLIB CALLS: None

▶ OPEN$A

OPEN$A is a logical function that opens a file of the given name on unit. If the operation is successful, the function value is .TRUE. and if the operation is unsuccessful, the function value is .FALSE..

```
LOG = OPEN$A(opnkey+typkey+untkey,name,namlen,unit)
CALL  OPEN$A(opnkey+typkey+untkey,name,namlen,unit)
```

opnkey A$READ, open for reading (.NE. A$WRIT or A$RDWR);
A$WRIT, open for writing;
A$RDWR, open for reading and writing.

typkey A$SAMF, SAM file (.NE. A$DAMF);
A$DAMF, DAM file.

untkey A$GETU, choose a file unit number to be returned in unit. Omission of this key requires that the routine be provided with a unit number.

name File name (may be a treename) packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2).

unit PRIMOS file unit (returned if untkey = A$GETU; if not, the caller must provide a legal file number in this argument). (INTEGER*2).

APPLIB CALLS: TREE$A, UNIT$A, NLEN$A

▶ OPNP$A

OPNP$A is a logical function that gets a name from the user and opens it on unit. If the operation is successful, the function value is .TRUE. and if the operation is unsuccessful or no name is supplied, the function value is .FALSE..

```
LOG = OPNP$A(msg,msglen,opnkey+typkey+untkey,name,namlen,unit)
CALL  OPNP$A(msg,msglen,opnkey+typkey+untkey,name,namlen,unit)
```

msg — Array containing prompt for name message, packed two characters per word. Data type does not matter.

msglen — Length of msg in characters (INTEGER*2).

opnkey — A$READ, open for reading
A$WRIT, open for writing
A$RDWR, open for reading and writing.

typkey — A$SAMF, SAM file
A$DAMF, DAM, file.

untkey — A$GETU, choose a file unit number to be returned in unit. Omission of this key requires that the routine be provided with a unit number.

name — Filename (may be a treename) packed two characters per word. Data type does not matter.

namlen — Length of name in characters (INTEGER*2).

unit — File unit (returned if untkey =A$GETU. If not, user must provide a legal file number in this argument). (INTEGER*2).

APPLIB CALLS: RNAM$A, NLEN$A, TREE$A, UNIT$A

▶ OPNV$A

OPNV$A is a logical function that opens a file of the given name on unit. Note that the functions of verification and delay as described below are independent of each other.

```
LOG = OPNV$A(opnkey+typkey+untkey,name,namlen,unit,verkey,wtime,retrys)
CALL  OPNV$A(opnkey+typkey+untkey,name,namlen,unit,verkey,wtime,retrys)
```

opnkey — A$READ, open for reading
A$WRIT, open for writing
A$RDWR, open for reading and writing.

typkey — A$SAMF, SAM file
A$DAMF, DAM file.

untkey — AGETU, choose a file unit number to be returned in unit. Omission of this key requires that the routine be provided with a unit number.

name Filename (may be a treename) packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2). If namlen is zero or less, the function value is .FALSE..

unit File unit (returned if untkey =A$GETU. If not, user must provide a legal file number in this argument). (INTEGER*2).

verkey A$NVER, no verification
A$VNEW, verify new or ask if ok to modify if old file.
A$OVAP, same as A$VNEW except user is prompted to "OVERWRITE" or "APPEND" if file already exists.
A$VOLD, verify old; return .FALSE. if not old file.

wtime Number of seconds to wait if FILE IN USE (INTEGER*2).

retrys Number of times to retry if FILE IN USE (INTEGER*2).

APPLIB CALLS: RNAM$A, TIME$A, NLEN$A, EXST$A, UNIT$A, TREE$A, GEND$A

If wtime and retrys are specified non-zero and the file to be opened is IN USE, the open will be retried the specified number of times, with wtime seconds (elapsed time) between each attempt. If the number of retries expires, or if either wtime or retrys is initially Ø and the file is IN USE, the function returns .FALSE.

Verification

If verification is not requested (VERKEY=A$NVER), OPNV$A is identical in function to OPEN$A. If verification is requested (verkey .NE. A$NVER), the following actions will be taken:

A$VNEW If the file already exists and opnkey is either A$WRIT or A$RDWR, the user will be asked if it is OK to modify the old file. If the answer is "NO", the function returns .FALSE.. If the answer is "YES", the file is opened.

A$OVAP This is the same as A$VNEW except that if an old file is to be modified, the user is also asked if the file should be overwritten or appended. If the answer is "append", the file will be positioned to End-of-File.

A$VOLD This is the default case if opnkey=A$READ. If any other key is specified, and if the named file does not already exist, a new file will not be created and the function returns .FALSE..

Errors

If any errors not covered above occur while opening the file or positioning it (A$OVAP), the function returns .FALSE.. If the open is ultimately successful, the function returns .TRUE..

▶ OPVP$A

OPVP$A is a logical function that gets a filename from the user and opens it on unit. Note that the functions of verification and delay as described below, are independent of each other.

```
LOG = OPVP$A(msg,msglen,opnkey+typkey+untkey,name,namlen,unit,
           verkey,wtime,retrys)
CALL  OPVP$A(msg,msglen,opnkey+typkey+untkey,name,namlen,unit,
           verkey,wtime,retrys)
```

msg — Array containing prompt message packed two characters per word. Data type does not matter.

msglen — Length of msg in characters (INTEGER*2).

opnkey — A$READ, open for reading
A$WRIT, open for writing
A$RDWR, open for reading and writing.

typkey — A$SAMF, SAM file
A$DAMF, DAM file.

untkey — AGETU, choose a file unit number to be returned in unit. Omission of this key requires that the routine be provided with a unit number.

name — Array containing filename (may be treename), packed two characters per word. Data type does not matter.

namlen — Length of name in characters (INTEGER*2). If namlen is zero or less, the function value is .FALSE..

unit — File unit (returned if untkey = A$GETU. If not, user must provide a legal file unit in this argument). (INTEGER*2).

verkey — A$NVER, no verification.
A$VNEW, verify new file or ask if OK to modify if old file.
A$OVAP, same as A$VNEW except user is prompted to "OVERWRITE" or "APPEND" if file already exists.
A$VOLD, verify old. Function value is .FALSE. if not old.

wtime Number of seconds to wait if FILE IN USE (INTEGER*2).

retrys Number of times to retry if FILE IN USE (INTEGER*2).

If wtime and retrys are specified non-zero and the file to be opened is IN USE, the open will be retried the specified number of times, with wtime seconds (elapsed time) between each attempt. If the number or retries expires, or if either wtime or retrys is initially Ø and the file is in use, the function returns .FALSE..

APPLIB CALLS: RNAM$A, TIME$A, NLEN$A, EXST$A, UNIT$A, TREE$A, GEND$A

Verification

If verification is requested, the following are the possible actions:

A$VNEW If the file already exists and opnkey is A$WRIT or A$RDR, the user will be asked if it is OK to modify the old file. If the answer is "NO", the function returns .FALSE.. If "YES", the file is opened.

A$OVAP If an old file is to be modified, (as answered "YES" for A$VNEW) the user is also asked if the file should be overwritten or appended. If the answer is "APPEND", the file will be positioned to End-of-File.

A$VOLD Default case if opnkey = A$READ. If any other key is specified, and if the named file does not already exist, a new file will not be created and the prompt message will be repeated.

Errors

If any errors not covered above occur while opening the file or positioning it (A$OVAP), or a name is not supplied when requested, the function returns .FALSE.. If the open is ultimately successful, the function returns .TRUE..

▶ POSN$A

POSN$A is a logical function which positions the file open on unit to the specified position. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = POSN$A(poskey,unit,pos)
CALL POSN$A(poskey,unit,pos)
```

poskey — A$ABS, absolute position (.NE. A$REL)
A$REL, relative position.

unit — PRIMOS file unit (INTEGER*2).

pos — Postion (relative or absolute) (INTEGER*4).

APPLIB CALLS: None

▶ RPOS$A

RPOS$A is a logical function which returns the current absolute position of the file open on unit. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = RPOS$A(unit,pos)
CALL  RPOS$A(unit,pos)
```

unit — PRIMOS file unit (INTEGER*2).

pos — Returned absolute position (INTEGER*4).

APPLIB CALLS: None

▶ RWND$A

RWND$A is a logical function that rewinds the file open on unit. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = RWND$A(unit)
CALL  RWND$A(unit)
```

unit — PRIMOS file unit (INTEGER*2).

APPLIB CALLS: None

▶ TEMP$A

This routine opens a unique temporary file in the current UFD for reading and writing. This file will be named T$xxxx where xxxx is a 4-digit decimal number between 0000 and 9999 inclusive. The actual name opened will be returned in the name buffer. If the operation is successful, the function value is .TRUE. and if the operation is unsuccessful, the function value is .FALSE..

```
LOG = TEMP$A(typkey+untkey,name,namlen,unit)
CALL  TEMP$A(typkey+untkey,name,namelen,unit)
```

typkey — A$SAMF, SAM file (.NE. A$DAMF)
A$DAMF, DAM file.

untkey — A$GETU, choose a file unit number to be returned in unit. Omission of this key requires that the routine be provided with a unit number.

name — Returned name (6 characters packed two characters per word). Data type does not matter.

namlen — Length of name buffer in characters (.GE. 6) (INTEGER*2).

unit — File unit (INTEGER*2).

APPLIB CALLS: FILL$A

▶ TRNC$A

TRNC$A is a logical function which truncates the file open on unit. If the operation is successful, the function is .TRUE. and if unsuccessful, the function is .FALSE..

```
LOG = TRNC$A(unit)
CALL  TRNC$A(unit)
```

unit — PRIMOS file unit (INTEGER*2).

APPLIB CALLS: None

▶ TSCN$A

TSCN$A is a logical function which scans the file system tree structure (starting with the home UFD) using the file subroutines RDEN$$ and SGDR$$ to read UFD and segment directory entries into the entry array.

LOG= TSCN$A(key,units,entry,maxsiz,entsiz,maxlev,lev,code)
CALL TSCN$A(key,units,entry,maxsiz,entsiz,maxlev,lev,code)

key	A$TREE, scan full tree A$NUFD, do not scan subufds A$NSEG, do not scan segment directories A$CUFD, scan current ufd only A$DLAY, pause when popping up to directory
units	Array of unit numbers maxlev long (INTEGER*2).
entry	Array maxsiz * maxlev long (INTEGER*2).
maxsiz	Size of each entry in entry array (INTEGER*2).
entsiz	Set to size of current entry (INTEGER*2).
maxlev	Maximum number of levels to scan (INTEGER*2).
lev	Current level (INTEGER*2).
code	Returned file system code (INTEGER*2).

APPLIB CALLS: None

Each call to TSCN$A returns the next file on the current level or the first file on the next lower level of the structure. The variable lev is used to keep track of the current level. For example, after the first call to TSCN$A (with lev=Ø), lev will be returned as 1, and entry(1,1) will contain the UFD entry describing the first file in the home UFD. If this file is a subufd, following the next call to TSCN$A, lev will be 2, and entry(1,2) will contain the entry for the first file in the subufd.

The values of key have the following meanings:

A$TREE	All entries in the tree structure are returned up to maxlev levels deep. (Levels below level maxlev are ignored.)
A$NUFD	When a subufd is encountered (in the home UFD), its entry is returned, but no files under that subufd are returned. In the absence of segment directories, this effectively limits the tree scan to the home UFD.
A$NSEG	Files inside segment directories are not returned.

A$CUFD This is a logical combination of A$NUFD and A$NSEG -- only files in the home UFD are returned.

A$DLAY This key is identical to A$TREE except that directory entries are returned twice, once on the way down (as for A$TREE), and again on the way up. (This is necessary, for example, to implement the tree-delete function since a directory cannot be deleted until it has been emptied.)

The following items should be considered when using TSCN$A:

1. For the first call of TSCN$A, lev should be equal to Ø. Thereafter it should not be modified until EOF is reached on the top level ufd at which point lev will be reset to Ø.

2. The entries in the entry array are in RDEN$$ format. For "entries" inside a segment directory, all information from the directory entry is first copied down a level. Entry(2,LEV) is set to Ø and entry(3,LEV) is then set to a 16-bit entry number. For nested segment directories, the type field of the entry is set appropriately by opening the file with SRCH$$. (The file is then immediately closed again.)

3. The parameter entsiz is set to the number of words returned by RDEN$$. Inside segment directories, it should be ignored.

4. The type fields in the entry array -- entry(2Ø,1) -- should not be modified. (TSCN$A uses them to walk up and down the tree.)

5. When TSCN$A requires a file unit, it uses units(lev). By using RDEN$$ and SGDR$$ read-position and set-position functions carefully, it is possible to dynamically reuse file units (e.g., to scan trees more than 16 levels deep.)

6. TSCN$A returns .TRUE. until a non-zero file system code is returned or until E$EOF is returned with lev=Ø (top level). E$EOF on lower levels of the tree is "suppressed", and code is returned as zero.

7. TSCN$A requires owner rights in the home UFD.

The following program illustrates how TSCN$A can be used to perform a tree LISTF:

```
$INSERT SYSCOM>ERRD.F
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>A$KEYS
C
      INTEGER MAXLEV,MAXSIZ
      PARAMETER MAXLEV=16 /* MAXIMUM LEVELS TO SCAN
      PARAMETER MAXSIZ=24 /* MAXIMUM SIZE OF EACH SLICE IN ENTRY
      INTEGER I,L,ENTRY(MAXSIZ,MAXLEV),UNITS(MAXLEV),CODE,NLEV$A
      LOGICAL TSCN$A
      DATA UNITS/1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16/
C
10    L=0  /* INITIALIZE LEVEL COUNTER
100   IF(TSCN$A(A$TREE,UNITS,ENTRY,MAXSIZ,I,MAXLEV,L,CODE))GOTO 105
      IF (CODE.NE.E$EOF) CALL ERRPR$(E$NRTN,CODE,0,0,0,0)
      CALL EXIT   /* ALL DONE IF E$EOF
      GOTO 10     /* RESTART IF 'S' TYPED
C
105   DO 200 I=1,L  /* CONSTRUCT TREENAME
        IF (ENTRY(2,I).EQ.0) GOTO 150  /* BRANCH IF SEGDIR
        CALL TNOUA(ENTRY(2,I),NLEN$A(ENTRY(2,I),32))
        GOTO 170
C
150     CALL TNOUA('(',1)  /* FORMAT SEGDIR ENTRY NUMBER
        CALL TODEC(ENTRY(3,I))
        CALL TNOUA(')',1)
C
170     IF (I.NE.L) CALL TNOUA(' > ',3)  /* TREENAME SEPARATOR
200     CONTINUE
      CALL TONL
      GOTO 100
      END
```

▶ UNIT$A

UNIT$A is a logical function which returns .TRUE. if the unit is open and .FALSE. if the unit is not open.

```
LOG = UNIT$A(unit)
```

unit PRIMOS file unit (INTEGER*2).

APPLIB CALLS: None

PARSING ROUTINE - DETAILED DESCRIPTION

The parsing routine is designed to facilitate the design and implementation of user interfaces in a program. It provides a means to break a character string into tokens and return information regarding each token.

▶ CMDL$A Parse PRIMOS type command line.

CMDL$A is a logical function for parsing a PRIMOS type command line and has the following calling sequence:

```
LOG = CMDL$A(key,kwlist,kwindx,optbuf,buflen,option,value,kwinfo)
CALL  CMDL$A(key,kwlist,kwindx,optbuf,buflen,option,value,kwinfo)
```

key A$READ, return the next keyword entry in the command line.

A$NEXT, call COMANL to get the next command line, turn on default processing, and return the first keyword entry in the new command line.

A$RSET, reset the command line pointer to the beginning of the command line and turn on default processing. Use of this key does not return a keyword entry.

A$RAWI, return the remainder of the command line as raw text and turn on the end of line indicator. Text starts at the token following the option (if present) of the last keyword entry read.

A$NKWL, turn on default processing and return the next keyword entry in the command line. This key allows the calling program to switch keyword lists in the middle of a command line.

A$RCMD, permits the use of a keyword without a preceding minus sign as the first token on a line (may only be used for lines subsequent to the initial command line).

kwlist A one dimensional array containing control information, a table of keyword entry descriptions, and a list of default keywords. See kwlist format later in this section for a complete description.

kwindx Keyword index, returned integer variable identifying the keyword in a keyword entry, possible values are:

< Ø, unrecognized keyword or CMDL$A was called with a key of A$RSET or A$RAWI.
=Ø, end of line.
> Ø, valid keyword.

optbuf Packed array that normally contains the text of a keyword option. However if an unrecognized keyword is encountered OPTBUF contains the text of that keyword. Data type does not matter.

buflen Specified length of optbuf in characters (INTEGER*2). (must be .GE. zero.)

option Option type, returned integer variable that describes the option following a keyword, possible values are:

A$NONE, no option, or option was null, optbuf will be blank.
A$NAME, option was a name
A$NUMB, option was a number, results of numeric conversion returned in value.
A$NOVF, option was a number and conversion resulted in overflow (decimal numbers only).

value Returned INTEGER*4 variable equal to the binary value of an option if it was a number. Otherwise it is zero.

kwinfo An array that returns miscellaneous information and must be dimensioned KWINFO(10) in the calling program. kwinfo(1) is equal to the number of characters in optbuf and kwinfo(2) - kwinfo(10) are reserved for future use.

APPLIB CALLS: CNVA$A, CNVB$A, CSUB$A, FILL$A, JSTR$A, MSUB$A, MSTR$A, NLEN$A, SSUB$A.

CMDL$A was designed to simplify the processing of a PRIMOS type command line while, at the same time, providing the user with a great deal of flexibility in defining his command environment.

This routine will parse a command line, a keyword entry at a time, and return information about each entry it encounters. A keyword entry is defined as a -keyword followed by an option. A default keyword entry is defined as an option that is not preceded by a -keyword but, by virtue of its position in the command line, implies a specified -keyword (e.g., FTN SNARF, where SNARF implies the default keyword -INPUT). Defaults may only occur at the beginning of a command line.

CMDL$A returns the following information for each keyword entry in the command line:

1. Integer that identifies the -keyword (kwindx).
2. Text of the keyword option, if present (optbuf).
3. Option type (option).
4. Results of numeric conversion, if option was a number (value).
5. Number of characters in the text of an option (kwinfo(1)).

Note that CMDL$A does not perform any action other than returning information about the command line.

The following is a list of considerations that should be taken into account when defining a command environment:

1. A keyword may have, at most, one option following it.
2. A keyword must begin with a '-'.
3. A keyword may not be a decimal number (e.g., -99).
4. Register setting parameters are not recognized as such.
5. Default keywords are only allowed at the beginning of a command line. The first -keyword encountered turns off default processing and all remaining options on the command line must be preceded by a -keyword (this restriction can be circumvented by using a key of A$NKWL, however the user must be aware of the fact that when default processing is in effect each option is treated as if it were preceded by a -keyword.
6. A key of A$RAWI (or an option type of A$RAWI) will turn on the end of line indicator and any further attempts to read from the current command line will return an end of line condition. To turn off the end of line indicator CMDL$A must be called with a key of A$RSET or A$NEXT.
7. A buffer length that is too small to contain the text of an option will cause that option to be truncated and an error message to be displayed.
8. Default keyword entries that have a numeric option should be avoided as PRIMOS may intercept them as register settings.
9. A negative hexadecimal option that consists of only alphabetic characters (e.g., -FF. will always be interpreted as a -keyword.

10. Keyword entries in the keyword table with the same keyword indicies are considered synonyms. A keyword may have any number of synonyms, each with different option specifications. However, if a keyword with synonyms is also a default and default processing is in effect, the option specifications for the synonyms will be ignored (i.e., a default keyword option always implies the first keyword in a synonym chain.

11. Null entries in the command line are only permitted for keywords that have an option status of A$OPTL. All other null entries will be treated as either a missing option or an unrecognized keyword.

12. Calls to CMDL$A and RDTK$$ on the same command line should be avoided, as CMDL$A uses RDTK$$ to perform a look-ahead when a -keyword is encountered.

13. All text is forced to upper case unless enclosed in quotes or read as raw text (A$RAWI).

KWLIST FORMAT

The kwlist array consists of three sections. The first section contains control information, the second contains the keyword entry table, and the third contains the default list.

CONTROL INFORMATION:

WORD 1 Number of keyword entries in table, must be .GT. zero.

WORD 2 Maximum length of keyword text in characters, must be .GE. 2 and .LE. 80. All keywords must have the same length and therefore it may be necessary to pad them with blanks.

KEYWORD TABLE:

Words 1 to n	Text of keyword, the actual number of characters must be equal to the maximum keyword length.
Word n+1	Keyword index, must be .GT. zero.
Word n+2	Minimum number of characters in the keyword to match including leading minus sign, must be .GE. 2 and .LE. maximum keyword length. A zero or negative value causes the keyword to be ignored when the table is searched. This allows keyword text to exist as documentation.
Word n+3	Option status, possible values are: A$NONE, no option may follow keyword, A$OPTL, option may or may not follow keyword, A$REQD, option must follow keyword.
Word n+4	Option type, possible values are: A$NONE, if status is A$NONE A$BIN, option must be a binary number A$DEC, option must be a decimal number A$OCT, option must be an octal number A$HEX, option must be a hexadecimal number A$NAME, option must be a name A$NBIN, option may be a name or binary number A$NDEC, option may be a name or a decimal number A$NOCT, option may be a name or an octal number A$NHEX, option may be a name or a hexadecimal number (if the option consists of all alphabetic characters, e.g., FACE, which also constitute a valid hexadecimal number, it will be interpreted as such). A$RAWI, option is the remainder of the command line after the current -keyword is read as raw text. Use of this option type will turn on the end of line indicator in the same manner as a key of A$RAWI.

Default List

Word 1	Number of default keywords, must be .GE. zero
Words 2 to n	(Where n is equal to word 1) list of keyword indices previously defined in the keyword entry table, that will be used when default processing is in effect. A default keyword entry may not have an option status of A$NONE.

Error Messages

The function value will be false if any of the following errors occur:

BAD KEY
BUFFER LENGTH LESS THAN ZERO
NAME TOO LONG. (name text)
UNRECOGNIZED KEYWORD. (keyword text)
BAD KEYWORD OPTION. (option text)
MISSING KEYWORD OPTION.
NO. OF KEYWORD ENTRIES MUST BE .GT. ZERO.
MAX KEYWORD LENGTH MUST BE .GE. 2 AND .LE. 8Ø.
1ST CHARACTER OF KEYWORD MUST BE '-'. (keyword text)
KEYWORD MAY NOT BE A NUMBER. (keyword text)
KEYWORD INDEX MUST BE .GT. ZERO. (keyword text)
MIN CHARACTERS TO MATCH MUST BE .LE. MAX KEYWORD LENGTH.
(keyword text)
INVALID OPTION STATUS. (keyword text)
INVALID OPTION TYPE. (keyword text)
NO. OF DEFAULTS MUST BE .GE. ZERO.
DEFAULT NOT DEFINED IN KEYWORD LIST. (default index)
INVALID DEFAULT OPTION STATUS. (keyword text)
MIN CHARACTERS TO MATCH MUST BE .GE. 2. (keyword text)
UNDETERMINED ERROR> (text of last keyword or option read)

CMDL$A Sample Program

```
C       TEST PROGRAM FOR CMDL$A
C
      IMPLICIT INTEGER*2 (A-Z)
      INTEGER*4 VALUE
      DIMENSION BUFFER(10),KWLIST(128),INFO(10)
$INSERT SYSCOM>A$KEYS
C
      DATA KWLIST /11,14,
     * '*any text     ',1,0,A$REQD,A$DEC,
     * '-NDECIMAL     ',2,2,A$OPTL,A$NDEC,
     * '-OCTAL        ',4,2,A$REQD,A$NONE,
     * '-NOCTAL       ',4,3,A$OPTL,A$NOCT,
     * '-HEXADECIMAL  ',5,2,A$REQD,A$HEX,
     * '-NHEXADECIMAL ',6,3,A$OPTL,A$NHEX,
     * '-NAME         ',7,5,A$REQD,A$NAME,
     * '-MAYBE        ',8,6,A$OPTL,A$NAME,
     * '-NONE         ',9,5,A$NONE,A$NONE,
     * '-QUIT         ',10,2,A$NONE,A$NONE,
     * '-TITLE        ',99,2,A$OPTL,A$RAWI,
     * 4,1,2,8,7/
C
C
      BUFLEN = 20
      KEY = A$READ
10    IF (CMDL$A(KEY,KWLIST,KWINDX,BUFFER,BUFLEN,TYPE,VALUE,INFO))
     *    GO TO 15
          PRINT 99
99        FORMAT(/'TRY AGAIN,TURKEY !')
          CALL EXIT
15    IF (KWINDX.EQ.10) CALL EXIT
      IF (KWINDX.NE.A$NONE) GO TO 20
          KEY = A$NEXT
          GO TO 10
20    KEY = A$READ
      PRINT 100 BUFFER,KWINDX,TYPE,VALUE,INFO(1)
100   FORMAT(/10A2/'KWINDX  TYPE  VALUE  CHARS'/2X,4(I3,6X))
      GO TO 10
      END
```

APPLIB SUMMARY AND KEYS

Below is a brief summary of the calling sequences for all the APPLIB routines and a listing of the file SYSCOM>A$KEYS.

In the summary that follows, the type codes are defined as:

```
L    = LOGICAL
I    = INTEGER (subject to compile time option)
I*2  = INTEGER*2
R    = REAL
DP   = DOUBLE PRECISION
```

Group	Name	Type	Arguments
File System	TEMP$A	L	(TYPKEY,NAME,NAMLEN,UNIT)
	OPEN$A	L	(OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN,UNIT)
	OPNP$A	L	(MSG,MSGLEN,OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN, UNIT)
	OPNV$A	L	(OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN,UNIT,VERKEY, WTIME,RETRYS)
	OPVP$A	L	(MSG,MSGLEN,OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN, UNIT,VERKEY,WTIME,RETRYS)
	CLOS$A	L	(UNIT)
	RWND$A	L	(UNIT)
	GEND$A	L	(UNIT)
	TRNC$A	L	(UNIT)
	DELE$A	L	(NAME,NAMLEN)
	EXST$A	L	(NAME,NAMLEN)
	UNIT$A	L	(UNIT)
	RPOS$A	L	(UNIT,POS)
	POSN$A	L	(POSKEY,UNIT,POS)
	TSCN$A	L	(KEY,UNITS,ENTRY,MAXSIZ, ENTSIZ,MAXLEV,LEV,CODE)

Group	Name	Type	Arguments
String	FILL$A	I	(NAME,NAMLEN,CHAR)
	NLEN$A	I*2	(NAME,NAMLEN)
	MCHR$A	I	(TARRAY,TCHAR,FARRAY,FCHAR)
	GCHR$A	I	(FARRAY,FCHAR)
	TREE$A	I	(NAME,NAMLEN,FSTART,FLEN)
	TYPE$A	L	(KEY,STRING,LENGTH)
	MSTR$A	I*2	(A,ALEN,B,BLEN)
	MSUB$A	I*2	(A,ALEN,AFC,ALC,B,BLEN,BFC,BLC)
	CSTR$A	L	(A,ALEN,B,BLEN)
	CSUB$A	L	(A,ALEN,AFC,ALC,B,BLEN,BFC,BLC)
	LSTR$A	L	(A,ALEN,B,BLEN,FCP,LCP)
	LSUB$A	L	(A,ALEN,AFC,ALC,B,BLEN,BFC,BLC,FCP,LCP)
	JSTR$A	L	(KEY,STRING,LENGTH)
	FSUB$A	L	(STRING,LENGTH,FCHAR,LCHAR,FILCHAR)
	RSTR$A	L	(STRING,LENGTH,COUNT)
	RSUB$A	L	(STRING,LENGTH,FCHAR,LCHAR,COUNT)
	SSTR$A	L	(STRING,LENGTH,COUNT,FILCHAR)
	SSUB$A	L	(STRING,LENGTH,FCHAR,LCHAR,COUNT,FILCHAR)
User Query	YSNO$A	L	(MSG,MSGLEN,DEFKEY)
	RNAM$A	L	(MSG,MSGLEN,NAMKEY,NAME,NAMLEN)
	RNUM$A	L	(MSG,MSGLEN,NUMKEY,VALUE)
Information	TIME$A	DP	(TIME)
	CTIM$A	DP	(CPUTIM)
	DTIM$A	DP	(DSKTIM)
	DATE$A	DP	(DATE)
	EDAT$A	DP	(EDATE)
	DOFY$A	DP	(DOFY)
Mathematical	RNDI$A	DP	(SEED)
	RAND$A	DP	(SEED)
Conversion	ENCD$A	L	(ARRAY,WIDTH,DEC,VALUE)
	CNVA$A	L	(NUMKEY,NAME,NAMLEN,VALUE)
	CNVB$A	I	(NUMKEY,VALUE,NAME,NAMLEN)
Parsing	CMDL$A	L	(KEY,KWLIST,KWINDX,OPTBUF,BUFLEN OPTION,VALUE,KWINFO)

SYSCOM>A$KEYS

```
C A$KEYS, APPLIB>SOURCE, ELS, Ø2/12/79
C Insert file for mnemonic APPLIB keys (FTN)
C Copyright 1977, PRIME COMPUTER, INC., Framingham, MA.
      NOLIST
C
C     ******************************************************************
C     *                                                                *
C     *              FUNCTION DECLARATIONS (TABSET 6 17)               *
C     *                                                                *
C     ******************************************************************
C
      LOGICAL   CLOS$A,RWND$A,GEND$A,TRNC$A,DELE$A,RPOS$A,POSN$A,TEMP$A,
     X          OPEN$A,OPNV$A,OPNP$A,OPVP$A,ENCD$A,YSNO$A,RNAM$A,RNUM$A,
     X          TREE$A,EXST$A,UNIT$A,CNVA$A,CMDL$A,CSUB$A,CSTR$A,TYPE$A,
     X          TSCN$A,JSTR$A,LSUB$A,LSTR$A,FSUB$A,SSTR$A,SSUB$A,RSTR$A,
     X          RSUB$A,CSP$$A,CASE$A
C
      INTEGER   MCHR$A,GCHR$A,FILL$A
      INTEGER*2 NLEN$A,MSUB$A,MSTR$A,CNVB$A
C
      REAL   *8 DOFY$A,DATE$A,EDAT$A,TIME$A,CTIM$A,DTIM$A,RNDI$A,RAND$A,
     X          FEDT$A,FTIM$A,FDAT$A
C
C     ******************************************************************
C     *                                                                *
C     *               KEY DECLARATIONS (TABSET 6 17)                   *
C     *                                                                *
C     ******************************************************************
C
      INTEGER*2 A$READ,A$WRIT,A$RDWR,A$SAMF,A$DAMF,A$NVER,A$VNEW,A$OVAP,
     X          A$VOLD,A$ABS ,A$REL ,A$DEC ,A$OCT ,A$HEX ,A$NDEF,A$DNO ,
     X          A$DYES,A$FUPP,A$UPLW,A$RAWI,A$NONE,A$OPTL,A$REQD,A$NDEC,
     X          A$NOCT,A$NHEX,A$NAME,A$NUMB,A$NEXT,A$RSET,A$RCMD,A$NKWL,
     X          A$NOVF,A$TREE,A$DLAY,A$NUFD,A$NSEG,A$CUFD,A$DECZ,A$DECU,
     X          A$OCTZ,A$HEXZ,A$RGHT,A$LEFT,A$CNTR,A$BACK,A$FLOW,A$BIN,
     X          A$NBIN,A$GETU,A$BINZ
C
      PARAMETER
     X
     X /*************************************************************** */
     X /*                                                               */
     X /*            KEY DEFINITIONS (TABSET 6 11 28 69)                */
     X /*                                                               */
     X /********************** OPEN$A ********************************** */
     X /********************** OPNP$A ********************************** */
     X /********************** OPNV$A ********************************** */
     X /********************** OPVP$A ********************************** */
     X /********************** TEMP$A ********************************** */
     X /*                ****** OPNKEY ******                           */
     X    A$READ = 1,       /* READ                                    */
```

```
X    A$WRIT = 2,       /* WRITE                                        */
X    A$RDWR = 3,       /* READ/WRITE                                   */
X /*                ****** TYPKEY ******                               */
X    A$SAMF = 0,       /* OPEN NEW SAM FILE                            */
X    A$DAMF = :2000,   /* OPEN NEW DAM FILE                            */
X /*                ****** UNTKEY ******                               */
X    A$GETU = :40000,  /* OPEN AND RETURN UNIT                         */
X /*                ****** VERKEY ******                               */
X    A$NVER = 1,       /* NO VERIFICATION                              */
X    A$VNEW = 2,       /* VERIFY NEW FILE (OK TO MODIFY)               */
X    A$OVAP = 3,       /* A$VNEW + OVERWRITE/APPEND OPTION             */
X    A$VOLD = 4,       /* VERIFY OLD FILE (DO NOT CREATE NEW)          */
X /*                                                                    */
X /********************* RPOS$A ********************************** */
X /*                ****** POSKEY ******                               */
X    A$ABS  = 1,       /* ABSOLUTE POSITION                            */
X    A$REL  = 2,       /* RELATIVE POSITION                            */
X /*                                                                    */
X /********************* YSNO$A ********************************** */
X /*                ****** DEFKEY ******                               */
X    A$NDEF = -1,      /* NO DEFAULT                                   */
X    A$DNO  = 0,       /* DEFAULT = 'NO'                               */
X    A$DYES = 1,       /* DEFAULT = 'YES'                              */
X /*                                                                    */
X /********************* RNUM$A ********************************** */
X /********************* CNVA$A ********************************** */
X /*                ****** NUMKEY ******                               */
X    A$DEC  = 1,       /* DECIMAL                                      */
X    A$OCT  = 2,       /* OCTAL                                        */
X    A$HEX  = 3,       /* HEXADECIMAL                                  */
X    A$BIN  = 9,       /* BINARY                                       */
X /*                                                                    */
X /*                                                                    */
X /********************* CNVB$A ********************                    */
X /*                ****** NUMKEY ******                               */
X /* A$DEC  = 1,      /* DECIMAL,LEFT PADDED WITH BLANKS              */
X /* A$OCT  = 2,      /* OCTAL, LEFT PADDED WITH BLANKS               */
X /* A$HEX  = 3,      /* HEXADECIMAL, LEFT PADDED WITH BLANKS         */
X /* A$BIN  = 9,      /* BINARY, LEFT PADDED WITH BLANKS              */
X    A$DECZ = 4,       /* DECIMAL, LEFT PADDED WITH ZEROS              */
X    A$OCTZ = 5,       /* OCTAL, LEFT PADDED WITH ZEROS                */
X    A$HEXZ = 6,       /* HEXADECIMAL, LEFT PADDED WITH ZEROS          */
X    A$DECU = 7,       /* UNSIGNED DECIMAL, LEFT PADDED WITH           */
X /*                       BLANKS                                       */
X    A$BINZ = 8,       /* BINARY, LEFT PADDED WITH ZEROS               */
X /*                                                                    */
X /*                                                                    */
X /********************* CMDL$A ********************************** */
X /*                ****** KEY    ******                               */
X /* A$READ = 1,      /* READ NEXT ENTRY IN COMMAND LINE              */
X    A$NEXT = 2,       /* READ FIRST ENTRY IN NEXT LINE                */
X    A$RSET = 3,       /* RESET TO BEGINNING OF COMMAND LINE           */
X /* A$RAWI = 4,      /* READ REMAINDER OF LINE AS RAW TEXT           */
```

```
X    A$NKWL = 5,      /* ACCEPT NEW KEYWORD LIST                    */
X    A$RCMD = 6,      /* FIRST TOKEN IS COMMAND (NO '-')            */
X /*                ****** OPTYPE ******                             */
X /* A$DEC  = 1,      /* DECIMAL OPTION                             */
X /* A$OCT  = 2,      /* OCTAL OPTION                               */
X /* A$HEX  = 3,      /* HEXADECIMAL OPTION                         */
X /* A$RAWI = 4,      /* OPTION IS RAW TEXT                         */
X    A$NDEC = 5,      /* NAME OR DECIMAL OPTION                     */
X    A$NOCT = 6,      /* NAME OR OCTAL OPTION                       */
X    A$NHEX = 7,      /* NAME OR HEXADECIMAL                        */
X    A$NAME = 8,      /* NAME                                       */
X /* A$BIN  = 9,      /* BINARY OPTION                              */
X    A$NBIN =10,      /* NAME OR BINARY OPTION                      */
X /*                ****** OPTION ******                             */
X    A$NONE = 0,      /* NO OPTION PRESENT OR NULL OPTION           */
X /* A$NAME = 8,      /* OPTION IS A NAME                           */
X    A$NUMB = 9,      /* OPTION IS A NUMBER (DIGIT STRING)          */
X    A$NOVF = 10,     /* NUMERIC OVERFLOW                           */
X /*                ****** STATUS ******                             */
X /* A$NONE = 0,      /* NO OPTION TO FOLLOW KEYWORD                */
X    A$OPTL = 1,      /* OPTION MAY OR MAY NOT FOLLOW KEYWORD       */
X    A$REQD = 2,      /* OPTION MUST FOLLOW KEYWORD                 */
X /*                                                                 */
X /********************* RNAM$A ********************************** */
X /*                ****** NAMKEY ******                             */
X    A$FUPP = 1,      /* FORCE UPPER CASE                           */
X    A$UPLW = 2,      /* READ UPPER AND LOWER CASE                  */
X    A$RAWI = 4,      /* READ REST OF LINE                          */
X /*                                                                 */
X /*                                                                 */
X /********************* TSCN$A ********************************** */
X /*                ****** KEY    ******                            */
X    A$TREE = 1,      /* ALL ENTRIES IN A TREE                      */
X    A$NUFD = 2,      /* DO NOT SCAN SUBUFDS                        */
X    A$NSEG = 3,      /* DO NOT SCAN SEGDIRS                        */
X    A$CUFD = 4,      /* DO NOT SCAN SUBUFDS OR SEGDIRS             */
X    A$DLAY = 5,      /* STAY AT DIRECTORY WHEN GOING UP TREE       */
X    A$BACK = 6,      /* BACK UP ONE LEVEL (FOR ERROR HANDLING)     */
X /*                                                                 */
X /********************* JSTR$A ********************************** */
X /*                ****** KEY    ******                            */
X    A$RGHT = 1,      /* RIGHT JUSTIFY                              */
X    A$LEFT = 2,      /* LEFT JUSTIFY                               */
X    A$CNTR = 3,      /* CENTER                                     */
X /*                                                                 */
X /********************* CASE$A ********************************** */
X /*                ****** KEY    ******                            */
X /* A$FUPP = 1,      /* FORCE UPPER CASE                           */
X    A$FLOW = 5       /* FORCE LOWER CASE                           */
X /*                                                                 */
X /********************* TYPE$A ********************************** */
X /*                ****** KEY    ******                            */
X /* A$BIN  = 9,      /* BINARY NUMBER                              */
```

```
X /* A$DEC  = 1,      /* DECIMAL NUMBER                                  */
X /* A$OCT  = 2,      /* OCTAL NUMBER                                    */
X /* A$HEX  = 3,      /* HEXADECIMAL NUMBER                              */
X /* A$NAME = 8       /* NAME                                            */
X /*                                                                     */
X /*********************************************************************/
 LIST
```

SECTION 12

SORT LIBRARIES

SORT SUBROUTINES OVERVIEW

PRIMOS contains many routines for performing disk or internal sorts. The subroutines are contained in the three libraries, described below. A detailed description of each subroutine follows later in this section.

SRTLIB is the R-mode library which contains two subroutines that call for a disk SORT operation.

VSRTLI is the V-mode version of the SRTLIB routines. These routines handle larger records and more data and record types than the R-mode version. VSRTLI also has subroutines which provide for the user's own input and output procedures.

MSORTS library contains several in-memory sort subroutines and also has a binary search/table building subroutine.

The following are the subroutines in each library.

SRTLIB	VSRTLI	MSORTS
SUBSRT	SUBSRT	HEAP
ASCS$$	ASCS$$	QUICK
	ASCSRT	SHELL
	SRTF$S	RADXEX
	SETU$S	INSERT
	RLSE$S	BUBBLE
	CMBN$S	BNSRCH
	RTRN$S	
	CLNU$S	
	MRG1$S	

Record Types

The following record types are handled by the sort library routines.

COMPRESSED SOURCE: Record with compressed blanks, delimited by a newline character (:212). Compressed source lines cannot contain data which may be interpreted as a blank compression indicator (:221) or newline character.

UNCOMPRESSED SOURCE: Record with no blank compression, delimited by a newline character (:212). Uncompressed source lines cannot contain data which may be interpreted as a newline character.

VARIABLE LENGTH: Record stored with length (in words) in the first word. This length does not include the first word which contains the count.

FIXED LENGTH: Record containing data but no length information. The length must be defined as the maximum line size. (If a newline character is appended to each record to make the file acceptable input to EDITOR, the character must be included in the length.)

Default depends upon the key-types specified (see Key Definitions, below). Input type defaults to variable length if a single precision integer, double precision integer, single or double precision real key is specified. Otherwise, the default is compressed source. If the output type is not specified, it is assumed to be the same as input type.

IF MULTIPLE INPUT FILES ARE USED THEY MUST ALL CONTAIN RECORDS OF THE SAME TYPE.

Record Length

The maximum record length allowed is 508 characters in R-mode and 32,760 characters in V-mode. "WARNING-LINE TRUNCATED" is printed whenever the data (not including record delimiters) exceeds the maximum record length and the excess data is ignored. Output record length defaults to the input record length.

Key Definitions

Each key must start and end on a byte boundary. An improperly defined key (e.g., record length is less than ending byte number of key) will produce indeterminate results. With compressed source records, the key is padded with spaces. In R-mode, 20 keys with a maximum length of 312 characters may be specified. In V-mode, up to 50 key fields may be specified and the total length may not exceed maximum record length. For fixed length records, key fields are verified to be within record length. The following are the key types which are specified as a parameter in the SORT subroutines.

ASCII keys are character strings, stored one character per byte. ASCII keys are limited only by the length of the record.

SIGNED NUMERIC ASCII keys require one byte per digit and include the following types:

Numeric ASCII, leading separate sign
Numeric ASCII, trailing separate sign
Numeric ASCII, leading embedded sign
Numeric ASCII, trailing embedded sign

A space will be treated as a positive sign. Signed numeric ASCII keys may be up to 63 digits plus sign.

When the sign is separate, a positive number has a plus sign(+) and a negative number has a minus sign(-). If the sign is embedded, a single character is used to represent the digit and sign. Embedded sign characters are:

Digit	Positive	Negative
Ø	Ø,-,+,{	};-
1	1 A	J
2	2 B	K
3	3 C	L
4	4 D	M
5	5 E	N
6	6 F	O
7	7 G	P
8	8 H	Q
9	9 I	R

UNSIGNED NUMERIC ASCII keys are stored one digit per byte and are limited only by the length of the record.

INTEGER and REAL keys include the following types:

KEY	BYTE LENGTH	RANGE
SINGLE PRECISION INTEGER	2	-32767 to +32767
DOUBLE PRECISION INTEGER	4	-2**31 to +2**31-1
SINGLE PRECISION REAL	4	±(1Ø**-38 to 1Ø**38)
DOUBLE PRECISION REAL	8	±(1Ø**-99Ø2 to 1Ø**9825)

PACKED DECIMAL keys are stored two nibbles (digit or sign) per byte. The last byte contains the final digit plus sign. A positive sign is represented by hex C in the sign nibble and a negative field has a hex D in the sign nibble. A packed field must have an odd number of digits and may have up to 63 digits plus sign.

SRTLIB (R-MODE) - SUBROUTINE DESCRIPTIONS

▶ SUBSRT

SUBSRT is used to sort a single input file, containing compressed source records, on ASCII keys in ascending order. Maximum record length is 508 characters and maximum key length is 312 characters.

CALL SUBSRT(tree-1,len-1,tree-2,len-2,numkey,nstart,nend,npass,nitem)

tree-1	Input treename.
len-1	Length of input treename in characters up to 80.
tree-2	Output treename.
len-2	Length of output treename in characters up to 80.
numkey	Number of pairs of starting and ending columns (maximum 20).
nstart	Vector containing starting columns of keys.
nend	Vector containing ending columns of keys.
npass	Number of passes (returned).
nitem	Number of items in output file (returned) INTEGER*4.

▶ ASCS$$

ASCS$$ is the R-mode subroutine which sorts or merges compressed or variable length records depending on the type of data specified in ntype. When sorting on binary files, starting and ending columns mean starting and ending bytes. When sorting equal keys, the input order is maintained. Maximum record length is 508 characters and maximum key length is 312 characters.

CALL ASCS$$ (tree-1,len-1,tree-2,len-2,numkey,nstart,nend,npass
nitem,nrev,ispce,mgcnt,mgbuff,len,LOC(buffer),msize
ntype,linsiz,nunits,units)

tree-1	Input treename.
len-1	Length of input treename in characters.

tree-2	Output treename.
len-2	Length of output treename in characters.
numkey	Number of pairs of starting and ending columns (maximum 20).
nstart	Vector containing starting columns.
nend	Vector containing ending columns.
npass	Number of passes (returned).
nitem	Number of items in output file (returned) INTEGER*4.
nrev	Vector containing order keys, 0=Ascending, 1=Descending.
ispce	0=sort blank lines, 1=delete blank lines.
mgcnt	Number of merge files (up to 10).
mgbuff	Array dimensioned (40*mgcnt) containing merge filenames.
len	Vector containing length of merge filenames in characters.
LOC(buffer)	Location of presort buffer in words.
msize	Size of presort buffer in words.
ntype	Vector containing type of each key.

Type	
1	ASCII
2	SINGLE PRECISION INTGER
3	SINGLE PRECISION REAL
4	DOUBLE PRECISION REAL
5	DOUBLE PRECISION INTEGER

linsiz	Maximum size of line in characters (default: 508 characters).
nunits	Number of file units available. (4 will be used)
units	Vector containing available file units.

Notes

1. Last 4 items are optional and may be omitted. Default value of ntype is ASCII.

2. Treenames may be used in ASCS$$ but may not exceed 80 characters in length.

3. Files specified as merge files will be sorted and merged with the input file. Treenames may be used for merge files, but only 10 merge files, no more than 80 characters in length may be used.

4. Presort buffer size should be as large as possible on P100 and P200 systems. On virtual memory systems, the best size must be determined by experimentation.

VSRTLI (V-MODE) - SUBROUTINE DESCRIPTIONS

VSRTLI routines follow a consistent naming convention to avoid possible conflict between user-written routines and system routines. All entry points end with the suffix $S - except SUBSRT and ASCSRT which remain the same for compatibility with earlier versions of the library. The names EB$1, EB$2, EB$3, EB$4, and EB$5 are no longer used. Subroutines used internally by VSRTLI routines which have a suffix of $$S should not be called from user routines. All parameters for all the routines are INTEGER*2 unless otherwise stated.

▶ SUBSRT

SUBSRT is used to sort a single input file, containing compressed source records, on ASCII keys in ascending order. Maximum record length is 32,760 bytes.

CALL SUBSRT(tree-1,len-1,tree-2,len-2,numkey,nstart,nend,npass,nitem)

tree-1	Input treename.
len-1	Length of input treename in characters, up to 80.
tree-2	Output treename.
len-2	Length of output treename in characters, up to 80.
numkey	Number of pairs of starting and ending columns, up to 50. If binary, specifies starting and ending bytes. Default = 1.
nstart	Vector containing starting columns/bytes(must be ≥1).

nend	Vector containing ending columns/bytes. Each ending column must be $\leq$ linsiz.
npass	Number of passes (returned).
nitem	Number of items in output file (returned) INTEGER*4.

▶ ASCSRT

ASCSRT (can also be called as ASCS$$ as in SRTLIB) is the V-mode subroutine which handles larger records and additional types of sort key fields, than the R-mode version. Maximum record length is 32,760 bytes.

ASCSRT sorts or merges compressed source or variable length records from and to disk files. Any of the supported key types (specified in ntype) may be used, and there may be ascending and descending keys within the same sort or merge. When sorting equal keys, the input order is maintained.

```
CALL{ASCS$$} (tree-1,len-1,tree-2,len-2,numkey,nstart,nend,npass
    {ASCSRT} nitem,nrev,ispce,mgcnt,mgbuff,len,LOC(buffer),msize
             ntype,linsiz,nunits,units)
```

tree-1	Input treename.
len-1	Length of input treename in characters, up to 80.
tree-2	Output treename.
len-2	Length of output treename in characters, up to 80.
numkey	Number of pairs of starting and ending columns, up to 50. If binary, specifies starting and ending bytes. Default = 1.
nstart	Vector containing starting columns/bytes. Each starting column must be $\geq$1.
nend	Vector containing ending columns/bytes. Each ending column must be $\leq$ linsiz.
npass	Number of passes (returned).
nitem	Number of items in output file (returned) INTEGER*4.

nrev — Vector containing sort order for each key.
Ø=Ascending, 1=Descending.
Default: Ø = Ascending

ispce — Option to specify treatment of blanks.
Ø = include blank lines in sort
1 = delete blank lines
Default: Ø = include blank lines

mgcnt — Number of merge files (up to 1Ø).

mgbuff — Array dimensioned (4Ø*mgcnt) containing merge filenames.

len — Vector containing length of merge filenames in characters, up to 8Ø.

LOC(buffer) — Location of presort buffer in words. For Rev. 17 and above, presort buffer is a common block, PSRT$S, and this parameter is ignored.

msize — Size (≤65536) of presort buffer in words. For Rev. 17 and above, corresponds to size of the common block, PSRT$S: This parameter may be a full 16-bit unsigned integer but cannot be INTEGER*4. If nonzero, msize must be at least 1Ø24 (one page) and no more than 64 pages. If larger, the message "WARNING-PRESORT BUFFER SHOULD NOT BE LARGER THAN ONE SEGMENT" is issued, and the default is used. Default: one segment (65536).

ntype — Vector containing type of each key.

Type	
1	ASCII
2	SINGLE PRECISION INTEGER
3	SINGLE PRECISION REAL
4	DOUBLE PRECISION REAL
5	DOUBLE PRECISION INTEGER

The additional types available for Rev.17 and above:

6	NUMERIC ASCII, LEADING SEPARATE SIGN
7	NUMERIC ASCII, TRAILING SEPARATE SIGN
8	PACKED DECIMAL
9	NUMERIC ASCII, LEADING EMBEDDED SIGN
1Ø	NUMERIC ASCII, TRAILING EMBEDDED SIGN
11	NUMERIC ASCII, UNSIGNED
12	ASCII, LOWER CASE SORTS EQUAL TO UPPER CASE

Default: ALL ASCII Keys.

linsiz — Maximum size of line in characters (bytes).
Default: 32768Ø

nunits — Number of file units available. (4 will be used) For Rev. 17 and above, parameter not used since file units are supplied dynamically using the subroutine SRCH$$ with key K$GETU (See Section 4).

units — Vector containing available file units. Obsolete for Rev. 17 and above. See nunits above.

Notes

1. Last 4 items are optional and may be omitted.
2. Files specified as merge files will be merged with the input file. Treenames may be used for merge files.
3. Presort buffer size should be determined by experimentation on virtual memory systems.

▶ SRTF$S

SRTF$S will sort input files (maximum 20) into a single output file.

```
CALL SRTF$S(inbuff,inlen,inunts,incnt,tree2,len2,outunt,
            numkey,nstart,nend,nrev,ntype,
            ercode,inrec,outrec,spcls,msize)
```

inbuff — Array dimensioned(40,incnt) containing input filenames.

inlen — Vector containing lengths of input treenames in characters, up to 80.

inunts — Vector containing input file units (if open units are used).

incnt — Number of input files (up to 20).

tree2 — Output file treename.

len2 — Length of output treename in characters, up to 80.

outunt — Output file unit (if an open unit is used).

numkey — Number of pairs of starting and ending columns (starting and ending bytes if binary), up to 50. Default = 1.

nstart — Vector containing starting columns/bytes. Each starting column must be ≥1.

nend Vector containing ending columns/bytes.
Each ending column must be ≤ maximum input line size.

nrev Vector containing sort order for each key
Ø = Ascending
1 = Descending
Default = Ø = Ascending.

ntype Vector containing type of each key
1 = ASCII
2 = SINGLE PRECISION INTEGER
3 = SINGLE PRECISION REAL
4 = DOUBLE PRECISION REAL
5 = DOUBLE PRECISION INTEGER
6 = NUMERIC ASCII, LEADING SEPARATE SIGN
7 = NUMERIC ASCII, TRAILING SEPARATE SIGN
8 = PACKED DECIMAL
9 = NUMERIC ASCII, LEADING EMBEDDED SIGN
1Ø = NUMERIC ASCII, TRAILING EMBEDDED SIGN
11 = NUMERIC ASCII, UNSIGNED
12 = ASCII, LOWER CASE, SORTS EQUAL TO UPPER CASE.
Default = ALL ASCII keys.

ercode Error code (returned).

inrec Five word array containing input record information:
inrec(1) = input record type
1 = COMPRESSED SOURCE (blanks compressed)
2 = VARIABLE LENGTH
3 = FIXED LENGTH (inrec(2) must be specified)
4 = UNCOMPRESSED SOURCE (no blank compression).
Default depends on the key types specified in argument ntype.
inrec(2) = Maximum input line size in characters (bytes).
Default = 32760.
Required for sorting fixed length records.
inrec(3-5) must be zero, and are reserved for future use.

outrec Five word array containing output record information.
outrec(1) = output record type (see inrec)
outrec(2) = maximum output line size in characters (bytes).
outrec(3-5) must be zero, and are reserved for future use.

spcls Five word array containing:
spcls(1) = Space option
Ø = include blank lines in sort
1 = delete blank lines
Default = Ø = include blank lines.
spcls(2-5) must be zero, and are reserved for future use.

msize Size of presort buffer in pages (units of 1024 words), ≤64. Note that the units used here are pages which differ from the words used by ASCSRT.
Default is one segment (64 pages).

▶ MRG1$S

MRG1$S merges two to eleven previously sorted files into a single output file.

```
CALL MRG1$S(inbuff,inlen,inunts,incnt,tree2,len2,outunt,numkey,
        nstart,nend,nrev,ntype,ercode,inrec,outrec,spcls,oproc)
```

inbuff — Array dimensioned (40,incnt) containing input filenames.

inlen — Vector containing lengths of input treenames in characters, up to 80.

inunts — Vector containing input file units(if open units are used)

incnt — Number of input files (up to 20).

tree2 — Output file treename.

len2 — Length of output treename in characters, up to 80.

outunt — Output file unit (if an open unit is used).

numkey — Number of pairs of starting and ending columns (starting and ending bytes if binary), up to 50. Default=1.

nstart — Vector containing starting columns/bytes. Each starting column must be ≥1.

nend — Vector containing ending columns/bytes. Each ending column must be ≤ inrec(2).

nrev — Vector containing sort order for each key
0 = Ascending
1 = Descending.
Default = 0 = Ascending.

ntype — Vector containing type of each key
1 = ASCII
2 = SINGLE PRECISION INTEGER
3 = SINGLE PRECISION REAL
4 = DOUBLE PRECISION REAL
5 = DOUBLE PRECISION INTEGER
6 = NUMERIC ASCII, LEADING SEPARATE SIGN
7 = NUMERIC ASCII, TRAILING SEPARATE SIGN
8 = PACKED DECIMAL
9 = NUMERIC ASCII, LEADING EMBEDDED SIGN
10 = NUMERIC ASCII, TRAILING EMBEDDED SIGN
11 = NUMERIC ASCII, UNSIGNED
12 = ASCII, LOWER CASE SORTS EQUAL TO UPPER CASE.
Default = ALL ASCII keys.

ercode — Error code (returned).

inrec — Five word array containing input record information:
inrec(1) = input record type
1 = COMPRESSED SOURCE (blanks compressed)
2 = VARIABLE LENGTH
3 = FIXED LENGTH (inrec(2) must be specified)
4 = UNCOMPRESSED SOURCE (no blank compression).
Default depends on the key types specified in ntype
inrec(2) = Maximum input line size in characters (bytes). Default = 32760. Required for sorting fixed length records.
inrec(3-5) must be zero, and are reserved for future use.

outrec — Five word array containing output record information:
outrec(1) = output record type (see inrec)
outrec(2) = maximum output line size in characters(bytes).
outrec(3-5) must be zero, and are reserved for future use.

spcls — Five word array containing:
spcls(1) = Space option
0 = include blank lines in sort
1 = delete blank lines.
Default = 0 = include blank lines.
spcls(2-5) must be zero, and are reserved for future use.

oproc — Output data destination
0 = Output file
1 = Output procedure.

SETU$S, RLSE$S, CMBN$S, RTRN$S, CLNU$S

The following five routines allow the user's own input and output procedures. These routines must all be called, and in the order given, to assure that the sort is done correctly. Source records passed to SORT from an input procedure must end with a newline character (:212). If not, a "WARNING-LINE TRUNCATED" message will be issued and the last character will be overwritten by a newline character. These subroutines are available in V-mode only. All parameters are INTEGER*2.

▶ SETU$S

SETU$S checks the parameters supplied by the user and sets up all tables for the particular sort being defined.

```
CALL SETU$S(inbuff,inlen,inunts,incnt,tree2,len2,outunt,
          numkey,nstart,nend,nrev,ntype,ercode,inrec,
          outrec,spcls,msize,iproc,oproc)
```

inbuff	Array dimensioned(40,incnt) containing input filenames.
inlen	Vector containing lengths of input treenames in characters, up to 80.
inunts	Vector containing input file units (if open units are used).
incnt	Number of input files (up to 20).
tree2	Output file treename.
len2	Length of output treename in characters, up to 80.
outunt	Output file unit (if an open unit is used).
numkey	Number of pairs of starting and ending columns (starting and ending bytes if binary), up to 50. Default = 1.
nstart	Vector containing starting columns/bytes, (must be $\geq$1).
nend	Vector containing ending columns/bytes, (must be $\leq$ inrec(2))

nrev Vector containing sort order for each key
Ø = Ascending
1 = Descending
Default = Ø = Ascending.

ntype Vector containing type of each key
1 = ASCII
2 = SINGLE PRECISION INTEGER
3 = SINGLE PRECISION REAL
4 = DOUBLE PRECISION REAL
5 = DOUBLE PRECISION INTEGER
6 = NUMERIC ASCII, LEADING SEPARATE SIGN
7 = NUMERIC ASCII, TRAILING SEPARATE SIGN
8 = PACKED DECIMAL
9 = NUMERIC ASCII, LEADING EMBEDDED SIGN
1Ø = NUMERIC ASCII, TRAILING EMBEDDED SIGN
11 = NUMERIC ASCII, UNSIGNED
12 = ASCII, LOWER CASE.SORTS EQUAL TO UPPER CASE.
Default = ALL ASCII keys.

ercode Error code (returned).

inrec Five word array containing input record information:
inrec(1) = input record type
1 = COMPRESSED SOURCE (blanks compressed)
2 = VARIABLE LENGTH
3 = FIXED LENGTH (inrec(2) must be specified)
4 = UNCOMPRESSED SOURCE (no blank compression).
Default depends on the key types specified in ntype.
inrec(2) = Maximum input line size in characters (bytes).
Required for sorting fixed length records.
Default = 32760.
inrec(3-5) must be zero, and are reserved for future use.

outrec Five word array containing output record information.
outrec(1) = output record type (see inrec)
outrec(2) = maximum output line size in characters (bytes).
outrec(3-5) must be zero, and are reserved for future use.

spcls Five word array containing:
spcls(1) = Space option
Ø = include blank lines in sort
1 = delete blank lines
Default = Ø = include blank lines.
spcls(2-5) must be zero, and are reserved for future use.

msize Size of presort buffer in pages (units of 1Ø24 words), ≤64.
Default is one segment (64 pages).

iproc Input data source
Ø = Input file
1 = Input procedure.

oproc Output data destination
Ø = Output file
1 = Output procedure.

▶ RLSE$S

RLSE$S transfers records to the initial phase of the sort. If an input procedure is used, RLSE$S is called once for each line released. If an input file is used, RLSE$S should be called only once.

CALL RLSE$S(rlbuff,length)

rlbuff Buffer containing next record for sort.

length Length of record in characters or bytes. This is not necessarily the full length of rlbuff.

If input is from a file, RLSE$S arguments are not used, and multiple calls to RLSE$S result in multiple occurrences of each record when sorted.

▶ CMBN$S

CMBN$S performs the internal sort. If the sort cannot be done within allocated memory, CMBN$S merges the strings previously sorted.

CALL CMBN$S

▶ RTRN$S

RTRN$S returns the sorted records. If an output procedure is used, each call to RTRN$S obtains the next sorted record. If an output file is specified, RTRN$S is called only once.

CALL RTRN$S(rtbuff,length)

rtbuff	Buffer containing next sorted record (returned). Should be large enough to hold the longest record sorted.
length	Length of record in characters or bytes (returned). When all records have been returned, calls to RLSE$S return a record length of Ø.

If output is to a file, RTRN$S arguments are not used.

▶ CLNU$S

CLNU$S closes all units opened by the sort routines and deletes any temporary files.

CALL CLNU$S

SAMPLE USER INPUT PROCEDURE

The following sample program demonstrates the use of an input procedure with the sort user-subroutines.

```
OK, SLIST SAMPLE
C-----SAMPLE PROGRAM WHICH CALLS SORT
C-----TO DEMONSTRATE THE USE OF AN INPUT PROCEDURE BEFORE SORTING
C
C
$INSERT SYSCOM>KEYS.F
$INSERT SYSCOM>ERRD.F
C
C
      INTEGER
     & BUFFER(10),                    /* Buffer for reading file
     & ERCODE,                        /* Error code
     & INREC(5),                      /* Input record information
     & OUTREC(5),                     /* Output record information
     & SPCLS(5),                      /* Flags for special options
     & TYPE                           /* File type returned when file
                                         opened
C
C
      DATA
C        Input records are fixed length (20 characters)
     &    INREC  / 3, 20, 0, 0, 0 /,
C        Output records are uncompressed source (so the file can be EDited)
     &    OUTREC / 4, 20, 0, 0, 0 /,
C        No special options
     &    SPCLS  / 0, 0, 0, 0, 0 /
C
C
C-----Open the input file
      CALL SRCH$$ (K$READ,'INPUTFILE',9,1,TYPE,ERCODE)
      IF (ERCODE .NE. 0) CALL ERRPR$(K$NRTN,ERCODE,0,0,0,0)
C
C-----Initialize sort tables
      CALL SETU$S
     &           (0,                  /* no input filenames
     &            0,                  /* no lengths of filenames
     &            0,                  /* no input file units
     &            0,                  /* no input filenames
     &            'OUTPUTFILE',       /* this is the output filename
     &            10,                 /* 'OUTPUTFILE' is 10 characters long
     &            0,                  /* no output file unit is specified
     &            1,                  /* sort file on one key
     &            1,                  /*   starting at column one
     &            20,                 /*   ending at column twenty
     &            0,                  /* sort in ascending order
     &            1,                  /* the key is all ASCII characters
     &            ERCODE,             /* an error code will be returned
```

```
     &            INREC,                  /* input record information
     &            OUTREC,                 /* output record information
     &            SPCLS,                  /* no special options requested
     &            0,                      /* use default value for presort buffer
     &            1,                      /* input data is from procedure
     &            0)                      /* output is to file.
      IF (ERCODE .NE. 0) CALL ERRPR$(K$NRTN,ERCODE,0,0,0,0)
C
C-----Read records from input file
100   READ (5,200,END=300) BUFFER
200   FORMAT (10A2)
C
C-----Select records to be sorted,
C-----   and pass them to sort with the record length (which is 20 characters)
      IF (BUFFER(1) .EQ. 'AA')  CALL RLSE$S (BUFFER,20)
      GO TO 100                          /* Go read next record
C
C-----Hit end of the input file, so finish up the sort
300   CALL CMBN$S                        /* do the sort
      CALL RTRN$S (0,0)                  /* output the records to the output file
      CALL CLNU$S                        /* clean up after sorting
C
C-----Close input file
      CALL SRCH$$ (K$CLOS,0,0,1,0,ERCODE)
      IF (ERCODE .NE. 0) CALL ERRPR$(K$NRTN,ERCODE,0,0,0,0)
      CALL EXIT
      END

OK, FTN SAMPLE -64V -DCLVAR
0000 ERRORS [<.MAIN.>FTN-REV17.0H

OK, SEG
[SEG rev 17.0
# LOAD #SAMPLE
$ LO B_SAMPLE
$ LI VSRTLI
$ LIB
LOAD COMPLETE
$ SAVE
$ QU

OK, SLIST INPUTFILE
AA   EMPLOYEE1
BB   EMPLOYEE5
BB   EMPLOYEE3
CC   EMPLOYEE4
AA   EMPLOYEE2
AA   EMPLOYEE6
CC   EMPLOYEE7
AA   EMPLOYEE0

OK, SEG #SAMPLE
```

```
OK, SLIST OUTPUTFILE
AA    EMPLOYEEØ
AA    EMPLOYEE1
AA    EMPLOYEE2
AA    EMPLOYEE6
```

MSORTS - SUBROUTINE DESCRIPTIONS

The MSORTS library contains several in-memory sort subroutines and a binary search/table building routine. The reference for most of the algorithm and timing studies is Donald Knuth, 'The Art of Computer Programming, Volume 3, Sorting and Searching'. It should be pointed out that the timing figures quoted are based upon Knuth's algorithms on his fictional machine (MIX). Since the MSORTS routines are more general, the Prime machines are different, and no in-house timing studies have yet been done, the timing formulas quoted here should be used only as an indication of the relative merits of each algorithm and not as exact computational tools.

In-memory sorts can be grouped into four categories: 1) sorts requiring very little additional memory; 2) sorts requiring a great deal of additional memory; 3) sorts using threaded lists; and 4) sorts based upon tree structures. Since available real memory space in Prime machines could be small and since the value of in-memory sorts using paged memory could be questionable, no sorts of categories 2 or 3 have been included in MSORTS. Also, since explicit tree structures tend to be threaded, only a single representative of this class has been included (HEAP).

The binary search routine (BNSRCH) can be used either for table lookup in an ordered table or for building a sorted table.

The routines included in MSORTS are:

HEAP	Heap Sort - based upon binary trees.
QUICK	'Quicksort' - partition-exchange.
SHELL	Shell Sort - diminishing increment.
RADXEX	Radix Exchange Sort.
INSERT	Straight Insertion Sort.
BUBBLE	'Bubble' Sort - interchange.
BNSRCH	Binary Search.

All routines accept multiword entries and multi-word keys located anywhere within the entry. The restrictions are that all entries are equal length and key words are contiguous (no secondary keys). An attempt has been made to keep the calling sequences as similar as possible. However, each sort has slightly different requirements. Except for RADXEX, all routines have the same first five parameters (arguments).

Parameters Common to More Than One Subroutine

ptable Integer pointer to the first word of the table. For example, if the table is in an array TABLE (a,b), the parameter ptable = loc (table). Ptable is a full 16 bit pointer and can be in the upper 32K of memory.

nentry Number of Integer table entries (not words) in the table (e.g., items to be sorted or searched). This is a full 16 bit count, since there can be more than 32K entries in the table.

nwds Number of words/entry. nwds > Ø. Obviously if nwds > 32K, there can be only a single entry.

fword First word within the entry of the key field.

nkwds Number of words in key field, Ø <nkwds ≤ nwds. Also, fword + nkwds - 1 ≤ nwds. (i.e., the key field must be contained within an entry).

npass Number of passes made (=Ø if error).

altbp Alternate return for bad parameters.

RADXEX replaces the nkwrds parameter by the following:

fbit First bit within fword of key. fbit > Ø and fword + (nbit + fbit -2)/16 ≤ nwds; i.e., the key field must be contained within an entry.

nbit Number of bits in key. nbit > Ø and fword + (nbit + fbit -2)/16 ≤ nwds; i.e., the key field must be contained within an entry.

Also, the routines HEAP, QUICK, RADXEX, and BUBBLE require temporary arrays of sizes:

HEAP,QUICK	tarray (nwds)
RADXEX	tarray (2nbit)
BUBBLE	tarray (nkwds)

All routines (except RADXEX) sort the table in increasing order where the key is treated as a single, signed multi-word integer. Therefore, the numbers 5, -1, 1Ø, -3 would be sorted to -3, -1, 5, 1Ø. RADXEX, since the key need not begin on a word boundary, treats the key as a single, unsigned multi-word (or partial word) integer. Thus, the same four numbers would be sorted by RADXEX to 5, 1Ø, -3, -1.

▶ BNSRCH

Simple binary searching (opflag=Ø) tests each entry's keyfield for a match with skey. If the entry is found, it is returned in fentry and the entry number is put into index. If the entry is not found, the not found alternate return (altnf) is taken. If altnf is not specified, the normal return is found, it is deleted from the table as well as returned in fentry. In this case, index specifies where the entry was. Opflag=2 is the same as opflag=Ø if the entry is found. If, however, the entry is not found, the contents of fentry will be inserted into the table and index will indicate the position of the new element. Also, altnf will be taken. Opflag=3 is the same as opflag=Ø if the entry is not found. If the entry is found, the contents of fentry and the found entry are interchanged, thus updating the table and returning the old entry.

```
CALL BNSRCH (ptable, nentry, nwds, fword, nkwds, skey, fentry,
             index, opflag, altnf, altbp)
```

The additional parameters are:

skey — Search key array (nkwds).

fentry — Found entry array (nwds).

index — Entry number of found entry.

opflag — operation key

Ø=locate
1=locate and delete
2=locate or insert
3=locate and update

altnf — Not found alternate return.

▶ BUBBLE

Bubble sorting is a simple interchange sort.

```
CALL BUBBLE (ptable, nentry, nwds, fword, nkwds, tarray, npass,
             altbp, incr)
```

incr — Same as in Insert; used to sort non-adjacent entries. Default = 1 (sort adjacent).

tarray — Must have nkwds words.

Running Time: The average running time is proportional to N**2. Bubble sorting is good only for very small N, but is not as good as insertion sorting.

▶ HEAP

Heap sort is based on a non-threaded binary tree structure. The algorithm consists of two parts: convert the table into a 'heap', and then sort the heap by an efficient 'top-down' search of the tree. The formal definition of a heap is:

> The Keys K(1), K(2), K(3),..., K(N) constitute a 'heap' if K(J/2)>K(J) for 1<J/2<J<N.

CALL HEAP (ptable, nentry, nwds, fword, nkwds, tarray, npass, altbp)

tarray Must have nwds words.

Running Time: The average running time is proportional to 23*N*lnN and the maximum is 26*N*lnN. Heap sort tends to be inefficient if N<2000, but for N>2000, it outperforms all other sorts except Quicksort.

▶ INSERT

Straight insertion sorting is based upon 'percolating' each element into its final position.

CALL INSERT (ptable, nentry, nwds, fword, nkwds, npass, altbp, incr)

incr Used to sort non-adjacent entries.

The incr parameter is used to sort non-adjacent entries. If, for example, incr=3, every third entry will be included in the sort. The default is incr=1. For example, with incr=3:

INPUT: 10 9 8 7 6 5 4 3 2 1 0
OUTPUT: 1 9 8 4 6 5 7 3 2 10 0

Running Time: Although the average running time is proportional to N**2, insertion sorting is very good for small tables (N<13) and tends to be very efficient for nearly ordered tables, even for large N.

▶ QUICKSORT

Quicksort is a partition exchange sort. QUICK is a variation of the basic quicksort called a median-of-three quicksort.

CALL QUICK (ptable, nentry, nwds, fword, nkwds, tarray, npass, altbp)

tarray Must have nwds words.

Running Time: The average running time is proportional to 12*N*lnN, but the maximum time is on the order of N**2. QUICK, on the average, is the fastest sort in MSORTS, but in the worst case, is about the slowest. In fact, the worst case is a completely ordered table. QUICK must not be used on tables that are already well ordered.

▶ RADXEX

RADXEX is a radix exchange sort that treats the key as a series of binary bits. It is based both on the method of radix sorting (like a card sorter) and partitioning. As noted before, RADXEX does not sort signed numbers and will sort the numbers 5, -1, 10, -3 to 5, 10, -3, -1. RADXEX has the advantage that the key need not start on a word boundary nor be an integral number of words long.

CALL RADXEX (ptable, nentry, nwds, fword, fbit, nbit, tarray, npass, altbp)

tarray Must have 2*nbit words; is used as partition stack.

Running Time: The average running time is proportional to 14*N*lnN. Radix exchange is very fast for large N (on the order of QUICK), but it is inefficient if equal keys are present.

▶ SHELL

SHELL sort (named after Donald Shell) is a diminishing increment sort. SHELL utilizes the straight insertion sort (INSERT) on each of its passes to order the non-adjacent elements an increment (which is decreased on each pass) apart. Increments are chosen by the formula:

Inc=(3**k-1)/2 where the initial increment is chosen so that inc(k + 2)>N and subsequent increments by decrementing k.

CALL SHELL (ptable, nentry, nwds, fword, nkwds, npass, altbp)

Running Time: The average running time is proportional to N**1.25 and the maximum time is N**1.5. A complete timing analysis on the SHELL sort is not possible, but for N<2000, it is very good. For N>2000, the HEAP SORT is better.

Source Files

All source files are in UFD=MSORTS on the master disk. These files are:

HEAP	Heap Sort
QUICK	Median-of-Three Quicksort
SHELL	Shell Sort
RADXEX	Radix Exchange Sort
INSERT	Straight Insertion Sort
BUBBLE	Bubble Sort
BNSRCH	Binary Search/Table Building

The following source files reside on the master disk and are called by the main library routines. They are not accessible by the user.

COMPAR
PERCOL
UCOMP
TSTBIT

Part IV
Input/Output
Library Subroutines

SECTION 13

INTRODUCTION TO IOCS

OVERVIEW OF IOCS

IOCS (the Input/Output Control System) is a group of subroutines that perform input/output between the Prime computer and the disks, terminals, and peripheral devices in the system. Generally, these functions may be grouped into three levels:

Level 1 Device-independent drivers (e.g., routines to read and write ASCII).

Level 2 Device specific drivers that issue the correct format for a particular device, but allow the output to be read later by device independent drivers.

Level 3 The lowest level of IOCS functions - routines to perform raw data transters (e.g., T$MT).

IOCS relates logical and physical devices so that callers of IOCS routines may be device-independent. The IOCS concept differs from the usual concept of logical and physical device that is of significance to the operating system.

Physical Devices

A physical device is a device type such as magnetic tape or a user terminal. Each device type is identified by physical device number as shown in table 13-1.

Table 13-1. Physical Devices Numbers

Physical Device	Device
1	User terminal
2	Paper-tape reader or punch
3	MPC card reader
4	Serial line printer
5	9-track magnetic tape ASCII/BINARY
6	MPC line printer
7	PRIMOS file system (compressed ASCII)
8	PRIMOS file system (uncompressed ASCII)
9	Serial card reader
10	7-track magnetic tape ASCII/BINARY
11	7-track magnetic tape BCD
12	(User terminal/command file)/command input
13	9-track magnetic tape EBCDIC
14	Versatec or Gould Printer/Plotter

Physical Unit

A physical unit designation distinguishes between the units of a physical device that has multiple units, such as a magnetic tape controller. For disk (file system), the physical unit corresponds to the file unit (FUNIT).

Logical Unit

The logical unit is the same as the unit number in FORTRAN READ and WRITE statements. IOCS translates and relates the physical device and logical units. The standard logical unit assignments are listed in Table 13-2. Table 13-3 shows logical-unit-to-physical-unit translation and Table 13-4 lists logical-unit-to-physical-device translation.

Table 13-2. Logical Device and Numbers

Logical Unit Number	Physical Device or Unit
1	User terminal
2	Paper-tape reader or punch
3	MPC card reader
4	Serial line printer
5	PRIMOS file unit 1
6	PRIMOS file unit 2
7	PRIMOS file unit 3
8	PRIMOS file unit 4
9	PRIMOS file unit 5
10	PRIMOS file unit 6
11	PRIMOS file unit 7
12	PRIMOS file unit 8
13	PRIMOS file unit 9
14	PRIMOS file unit 10
15	PRIMOS file unit 11
16	PRIMOS file unit 12
17	PRIMOS file unit 13
18	PRIMOS file unit 14
19	PRIMOS file unit 15
20	PRIMOS file unit 16
21	9-track magnetic tape unit 0
22	9-track magnetic tape unit 1
23	9-track magnetic tape unit 2
24	9-track magnetic tape unit 3
25	7-track magnetic tape unit 0
26	7-track magnetic tape unit 1
27	7-track magnetic tape unit 2
28	7-track magnetic tape unit 3

Table 13-3. Logical-Unit-To-Physical-Unit Translation

```
*       PUTBL---LOGICAL UNIT=> PHYSICAL UNIT TRANSLATION TABLE
*
*
*
PUTBL   DEC     Ø           Ø1 => ØØ
        DEC     Ø           Ø2 => ØØ
        DEC     Ø           Ø3 => ØØ
        DEC     Ø           Ø4 => ØØ
        DEC     1           Ø5 => Ø1
        DEC     2           Ø6 => Ø2
        DEC     3           Ø7 => Ø3
        DEC     4           Ø8 => Ø4
        DEC     5           Ø9 => Ø5
        DEC     6           1Ø => Ø6
        DEC     7           11 => Ø7
        DEC     8           12 => Ø8
        DEC     9           13 => Ø9
        DEC     1Ø          14 => 1Ø
        DEC     11          15 => 11
        DEC     12          16 => 12
        DEC     13          17 => 13
        DEC     14          18 => 14
        DEC     15          19 => 15
        DEC     16          2Ø => 16
        DEC     Ø           21 => Ø
        DEC     1           22 => 1
        DEC     2           23 => 2
        DEC     3           24 => 3
        DEC     Ø           25 =>.Ø
        DEC     1           26 => 1
        DEC     2           27 => 2
        DEC     3           28 => 3.
PUTBLE  EQU     *
*
*
*
```

Table 13-4. Logical-Unit-To-Physical-Device Translation

```
*       LUTBL --- LOGICAL UNIT => PHYSICAL DEVICE TRANSLATION TABLE
*
*
        LINK
*
LUTBL   DEC     1           Ø1 => Ø1
        DEC     2           Ø2 => Ø2
        DEC     3           Ø3 => Ø3
        DEC     4           Ø4 => Ø4
        DEC     7           Ø5 => Ø7
        DEC     7           Ø6 => Ø7
        DEC     7           Ø7 => Ø7
        DEC     7           Ø8 => Ø7
        DEC     7           Ø9 => Ø7
        DEC     7           1Ø => Ø7
        DEC     7           11 => Ø7
        DEC     7           12 => Ø7
        DEC     7           13 => Ø7
        DEC     7           14 => Ø7
        DEC     7           15 => Ø7
        DEC     7           16 => Ø7
        DEC     7           17 => Ø7
        DEC     7           18 => Ø7
        DEC     7           19 => Ø7
        DEC     7           2Ø => Ø7
        DEC     5           21 => Ø5
        DEC     5           22 => Ø5
        DEC     5           23 => Ø5
        DEC     5           24 => Ø5
        DEC     1Ø          25 => 1Ø
        DEC     1Ø          26 => 1Ø
        DEC     1Ø          27 => 1Ø
        DEC     1Ø          28 => 1Ø
LUTBLE  EQU     *
*
*
*
```

TEMPORARY DEVICE ASSIGNMENT

The user may assign any device by calling the ATTDEV subroutine. ATTDEV controls mapping of logical units into physical devices and controls the record size associated with the logical unit. Non-shareable devices are assigned on command level with the PRIMOS 'AS' command. When a permanent device assignment is desired the reader should go on to the CONIOC description.

▶ ATTDEV

ATTDEV attaches specified devices by initializing both LUTBL, associating logical-device to physical-device, and PUTBL, associating the logical device to a specific unit or file of the device.

CALL ATTDEV (logical-device, physical-device, unit, buffer-size)

logical-device	The device-independent logical I/O unit for WRASC, RDASC, WRBIN, RDBIN, and FORTRAN READ and WRITE statements.
physical-device	The position in the device-type tables.
unit	The unit for multi-unit devices (e.g., for disk, file unit number).
buffer-size	The record size associated with the logical unit. Must be as large as maximum record size.

For the given logical-device, set the physical-device, unit, and buffer-size in the LUTBL, PUTBL, and RSTBL so that the logical unit has a current mapping.

Errors

If device is incorrect, ATTDEV returns the message: ATTDEV BAD UNIT (unit-number).

CONIOC

To facilitate changes to device assignments, the tables used by IOCS (such as LUTBL and PUTBL) are in a file named CONIOC in a UFD named IOCS for R-mode and IOCSV for V-mode. The user should list these files on his system for reference before making any changes.

Note that the R-mode CONIOC in the FTNLIB supports only the user terminal, the paper-tape reader, paper-tape punch, and the PRIMOS file system. An attempt to perform I/O to a physical-device not supported by CONIOC will fail. The default CONIOC for V-mode supports the user terminal and PRIMOS file system only.

Users who consider that their programs need to use devices other than the user terminal, the disks, or paper-tape reader or punch, must refer to the discussion of IOCS tables which follows. Users who wish to change the asignment of logical to physical devices must also refer to the IOCS tables.

IOCS Tables

If a computer installation requires that user programs use devices not supported by CONIOC, the system administrator must modify the CONIOC tables RATBL, RBTBL, WATBL, and WBTBL, and then rebuild the FORTRAN library. File FULCON (in the UFD named IOCS for R-mode, IOCSV for V-mode) is a version of CONIOC that contains all the available IOCS drivers set up in the appropriate tables. The user should list this file for reference before making any changes. The operator can use FULCON as an example of how to set up CONIOC. The entries in the tables that are not required can be set to zero.

The operator may also change the default logical-to-physical-device association as given in Tables 13-1 and 13-2 by changing the IOCS tables RATBL, TBTBL, WATBL, and CNTBL. For example, the fifth entry of LUTBL (indicating logical device 5) contains 7. Entry 7, the RATBL, contains I$AD07, which is a driver for the PRIMOS file system. Other numbers indicate physical devices, as shown in Table 13-1. PUTBL is the sub-unit table. The sub-unit table contains the individual unit or file numbers as requird for multi-file devices. For example, LUTBL contains the same number of logical devices 21, 22, 23, and 24, indicating 9-track magnetic tape. PUTBL contains 0, 1, 2, and 3 for logical devices 21, 22, 23, and 24 indicating ulnit 0, 1, 2, and 3 of 9"-track magnetic tapes.

Modifying CONIOC To Change Device Assignment

Changing a device assignment is a system administrator responsibility and not a user function, Thus, the system administrator may add or delete a device to:

RATBL	Read ASCII table
RBTBL	Read Binary table
WATBL	Write ASCII table
WBTBL	Write Binary table
CNTBL	perform control function (e.g., endfile, rewind, etc.)

Input Only Devices

Input only devices (e.g., card reader) do not need WATBL and WBTBL entries. Furthermore, an ASCII only device (e.g., line printer) does not need RBTBL and WBTBL entries.

Order of Entries

The order of entries in the above mentioned tables correspond to physical device numbers defined in Table 13-1.

R-Mode Procedures

Step 1: Attach to IOCS of Master disk A.

Step 2: Edit the appropriate tables within the CONIOC.

Step 3: Replace the 0 with the corresponding Subroutine name for the desired device.

Step 4: Rebuild the FORTRAN Library.

V-Mode Procedures

Step 1: Attach to IOCSV of Master Disk A.

Step 2: Edit the appropriate tables within the CONIOC.

Step 3: Replace the word NULLDEVICE with the appropriate device subroutine name.

Step 4: Rebuild the FORTRAN Library.

How To Rebuild The FORTRAN Library After Modifying CONIOC

The FORTRAN Library must be rebuilt whenever CONIOC is modified. The following explanations are R-Mode and V-Mode procedures.

R-Mode FORTRAN Library Re-building Procedures

The R-Mode FORTRAN Library must be remade after CONIOC has been modified:

Step 1: Attach to UFD = IOCS, in Master Disk A.

Step 2: Run Command File C<IOCS.

Step 3: Attach to UFD = LIB, in Master Disk A.

Step 4: Run Command File LIBMAK.

V-Mode FORTRAN Library Re-building Procedures

The V-Mode FORTRAN Library must be remade after CONIOC has been modified:

Step 1: Attach to UFD = IOCSV, in Master Disk A.

Step 2: Run Command File BNIOCS, in Master Disk A.

Step 3: Attach to UFD = LIB, in Master Disk A.

Step 4: Run Command File C_VLIB.

SECTION 14

I-O SUBROUTINES

This section describes input/output subroutines that reside in PRIMOS address space, rather than user address space, but are directly callable by the user.

▶ D$INIT

The D$INIT routine is called to initialize disk devices.

CALL D$INIT (pdisk)

pdisk The physical disk number to be initialized.

D$INIT initializes the disk controller and performs a seek to cylinder Ø on pdisk. D$INIT must be called prior to any RRECL or WRECL calls. pdisk must be assigned by the PRIMOS ASSIGN command before calling this routine. D$INIT is normally used only by system utilities such as FIXRAT, COPY, and MAKE.

▶ RRECL

Subroutine RRECL reads one disk record from a disk into a buffer in memory. Before RRECL is called, the disk must be assigned by the PRIMOS ASSIGN command and D$INIT must be called to initialize the disk.

The RRECL routine is normally used only by system utilities such as FIXRAT, MAKE, and COPY.

CALL RRECL (LOC(buffer), length, n, ra, pdisk, altrtn)

buffer An array into which the length words from record ra will be transferred.

length The number of words to be transferred.

n Bits 9-16 must be 1.

Bit 1 set- perform current record address check.

Bit 2 set- ignore checksum error.

Bit 3 set- read an entire track (beginning at ra) into a buffer 3520 words long, beginning at the buffer pointed to by buffer. (This feature may be used only if RRECL is running under PRIMOS II and is reading a disk connected to the 4001/4002 controller and is a 32-sector pack.)

Bit 4 set- format the track. This bit is only significant for storage module disks.

ra — A 32 bit integer (INTEGER*4) specifying a disk record address. Legal addresses depend on the size of the disk.

Size	ra Range
Floppy disk	0-303
1.5M disk pack	0-3247
3.0M disk pack	0-6495
30M disk pack	0-64959
128K fixed-head disk	0-255
256K fixed-head disk	0-511
512K fixed-head disk	0-1023
1024K fixed-head disk	0-2047

pdisk — The physical disk number of the disk to be read. pdisk numbers are the same numbers available for use in the ASSIGN and STARTUP commands.

altrtn — An integer variable in the user's program to be used as an alternate return in case of uncorrectable disk errors. If this argument is 0 or omitted, an error message is printed.

If an error is encountered and control goes to altrtn, ERRVEC is set as follows:

Code	Message	Meaning
ERRVEC(1) = WB ERRVEC(2) = 0	On supervisor terminal: 10 times DISK RD ERROR pdisk ra status On user terminal: UNRECOVERED ERROR	Disk hardware WRITE PROTECT error
ERRVEC(1) = WB ERRVEC(2) = CR	On user terminal: 10 times DISK RD ERROR pdisk ra status followed by UNRECOVERED ERROR	Current record address error

See The System Administrator's Guide (PDR 3109) for a description of status error codes.

Notes

Length must be between Ø and 448 unless pdisk is a storage module, in which case length must be between Ø and 1040. If this number is not 448 and pdisk is 20-27 (diskette), a checksum error is always generated; bypassing can be accomplished by setting n bit 2 = 1. No check is made for legality of ra.

On a DISK NOT READY, RRECL does not wait for the disk to become ready under PRIMOS III or PRIMOS. Under PRIMOS II, RRECL prints a single error message and waits for the disk to become ready.

On any other read error, an error message is printed at the system terminal, followed by a seek to cylinder zero and a reread of the record. If 10 errors occur, the message UNRECOVERED ERROR is typed to the user or altrtn is taken.

The routine is not available through the FORTRAN library.

▶ WRECL

Subroutine WRECL writes the disk record to a disk from a buffer in memory. The arguments and rules of the WRECL call are identical to those of RRECL except for bits 1 and 2 of n, which have no meaning on write. For a call to write a record on the diskette, the buffer length must be 448 words.

CALL WRECL (LOC(buffer), length, n, ra, altrtn)

The meaning of the parameters is the same as described above in RRECL, except that the function of the command is to write rather than read the specified records. The user of WRECL is responsible for being careful to write only on areas of the disk that do not contain significant user or operating system information.

An attempt to write on a write-protected disk generates the message:

```
DISK WT ERROR    pdisk    ra    status
WRITE PROTECT
```

on the supervisor terminal and the message:

```
UNRECOVERED ERROR
```

at the user terminal. ERRVEC(1) will contain error code WB, unless altrtn is taken. Other write errors are retried ten times in a manner similar to read errors (refer to RREC). This routine is not available through the FORTRAN library.

ERROR HANDLING FOR I-O SUBROUTINES

The following discusses error handling for the I/O subroutines. Generally, error message and status information from PRIMOS I/O subroutines, and some older PRIMOS routines, are placed in a system-wide error vector, ERRVEC. If an error occurs, the user program returns to PRIMOS command level and the error and/or status information is placed in ERRVEC. Upon completion of a call to an I/O subroutine, status information is also placed in ERRVEC, which the user may acces via a call to GINFO or PRERR. The contents of this vector are listed later in this section. If the user so desires, it is possible to take an alternate return if an error occurs. This is specified by use of the altrtn parameter in the call to the I/O subroutine invoked by the user program. If the user specifies alternate return then the location of the return and the action taken is entirely up to the user.

Subroutines for Error Return and Printing

Three subroutines are useful for setting or retreiving information in ERRVEC: ERRSET, GETERR, PRERR.

▶ ERRSET

ERRSET sets ERRVEC, a system vector, then takes an alternate return or prints the message stored in ERRVEC and returns control to the system.

ERRSET has three forms:

1. CALL ERRSET (altval, altrtn)
2. CALL ERRSET (altval, altrtn, messag, num)
3. CALL ERRSET (altval, altrtn, name, messag, num)

In Form 1, altval must have value 100000 octal and altrtn specifies where control is to pass. If altrtn is 0, the message stored in ERRVEC is printed and control returns to the system.

Forms 2 and 3 are similar; therefore, the arguments are described collectively as follows:

altval	A two-word array that contains an error code that replaces ERRVEC(1) and ERRVEC(2). altval(1) must not be equal to 100000 octal.
altrtn	If altrtn is nonzero, control goes to altrtn. If altrtn is zero, the message stored in ERRVEC, is printed and control returns to PRIMOS.

name	The name of a three-word array containing a six-letter word. This name replaces ERRVEC(3), ERRVEC(4), and ERRVEC(5). If name is not an argument in the call, ERRVEC(3) is set to Ø.
messag	An array of characters stored two per word. A pointer to this messag is placed in ERRVEC(7).
num	The number of characters in messag. The value of num replaces ERRVEC(8).

If a message is to be printed; first, six characters starting at ERRVEC(3) are printed at the terminal. Then the operating system checks to determine the number of characters to be printed. This information is contained in ERRVEC(8). The message to be printed is pointed to by ERRVEC(7). The operating system only prints the number of characters from the message (pointed to by ERRVEC(7)) that are indicated in ERRVEC(8). If ERRVEC(3) is Ø, only the message pointed to by ERRVEC(7) is printed. The message stored in ERRVEC may also be printed by the PRERR command or the PRERR subroutine. The contents of ERRVEC may be obtained by calling subroutine GETERR.

▶ GETERR

A user obtains ERRVEC contents through a call to GETERR.

CALL GETERR (xervec, n)

GETERR moves n words from ERRVEC into xervec.

On an alternate return:

ERRVEC(1) Error code

ERRVEC(2) Alternate value

On a normal return:

PRWFIL:
ERRVEC(3) Record number
ERRVEC(4) Word number

Key of read/write convenient:
ERRVEC(2) No. of words transferred

SEARCH:
ERRVEC(2) File type

▶ PRERR

PRERR prints an error message on the user's terminal.

CALL PRERR

Example of Use

A user wants to retain control on a request to open a unit for reading if the name was not found by SEARCH. To accomplish this, the user calls SEARCH and gets an alternate return. He then calls to GETERR and determines if an error occurred other than NAME NOT FOUND. To print the error message while maintaining program control, the user calls PRERR.

Description of ERRVEC

ERRVEC consists of eight words; their contents are as follows:

Word	Content	Remarks
ERRVEC (1)	Code	Indicates origin of error and nature of error.
(2)	Value	On alternate return, this is the value of the A-register. On normal return, this may have special meaning, (e.g., refer to PRWFIL and SEARCH error codes).
(3)	X X	ERRVEC (3), ERRVEC (4), ERRVEC (5), and ERRVEC (6) contain a six-character Filename of the routine that caused the error [ERRVEC (6) is available for expansion of names].
(4)	X X	
(5)	X X	
(6)	X X	
(7)	Pointer To Message	For PRIMOS supervisor usage.
(8)	Message Length	For PRIMOS supervisor usage.

PRWFIL Error Codes

PD	UNIT NOT OPEN	
PE	PRWFIL EOF (End of File)	Number of words left. (Information is in ERRVEC(2))
PG	PRWFIL EOF (Beginning of File)	Number of words left. (Information is in ERRVEC(2))

PRWFIL Normal Return

ERRVEC (3) Record Number

ERRVEC (4) Word Number

PRWFIL Read-Convenient

ERRVEC (2) Number of words read.

SEARCH Error Codes

ERRVEC (1) Code, where code has the following values:

Code	Meaning
SA	SEARCH, BAD PARAMETER
SD	UNIT NOT OPEN (truncate)
SD	UNIT OPEN ON DELETE
SH	<Filename> NOT FOUND
SI	UNIT IN USE
SK	UFD FULL
SL	NO UFD ATTACHED
SQ	SEG-DIR-ER
DJ	DISK FULL

SEARCH Normal Return

ERRVEC (2) Type, where Type has the following values:

Type	Meaning
Ø	File is SAM
1	File is DAM
2	Segment Directory is SAM
3	Segment Directory is DAM
4	UFD is SAM

SECTION 15

DEVICE INDEPENDENT DRIVERS

To maintain device independence, all data transfer is accomplished through a set of device-independent drivers in IOCS. These device-independent drivers route the I/O request to one of the device-dependent drivers, as shown in Table 15-1 and Figure 15-1. Each column of this table represents an I/O function, and each row a specific physical device. All drivers in a single column are designed to be compatible in terms of internal data format.

Notes to Table 15-1

1. Available in R-mode and V-mode. Listed in CONIOC and may be called directly or via the device-independent drivers.

2. Available in R-mode only. Listed in CONIOC and may be called directly or via the device-independent drivers.

3. Available in R-mode only. Listed in FULCON but not CONIOC. May not be called via the device-independent drivers, unless FULCON is assembled and loaded before the library is loaded.

4. Available in R-mode and V-mode. Listed in FULCON (FLCONV for V-mode). In V-mode programs, these routines may be called directly or via the device-independent drivers if the default FORTRAN library (PFTNLB) is loaded. If the R-mode or the non-shared V-mode library (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers unless FLCONV or FULCON is assembled and loaded before the library is loaded. See Section 13 for a more complete discussion of IOCS table usage. Routine may be called by name without specific procedures.

5. Available in R-mode and V-mode. For R-mode, is listed in CONIOC and may be called directly or via the device independent drivers. For V-mode, routine is listed in FLCONV and may be used in same manner as R-mode as long as the default FORTRAN library (PFTNLB) is loaded. In R-mode, or V-mode when the non-shared (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers drivers unless FULCON (FLCONV) is assembled and loaded before the library is loaded. See Section 13 for a more complete discussion of IOCS table usage.

6. Available in R-mode and V-mode, but is not in CONIOC or FULCON (FLCONV). To call the routines via the device independent drivers, the appropriate table must be modified, assembled and loaded before the library is loaded. See Section 13. The routine may be called specifically without any special procedures.

Table 15-1. Relation of Device-Independent and Device-Dependent Drivers

Device	RDASC	WRASC	RDBIN	WRBIN	CONTRL
User terminal	I$AAØ1(6)	O$AAØ1(1)	I$BAØ1(2)	O$BAØ1(2)	C$AØ1(2)
Paper tape reader	I$APØ2(5)		I$BPØ2(2)		C$PØ2(5)
Paper tape punch		O$APØ2(5)		O$BPØ2(2)	
MPC card reader	I$ACØ3(3)	O$ACØ3(3)			
Serial line printer		O$ALØ4(3)			
9-track mag.tape	I$AMØ5(4)	O$AMØ5(4)	I$BMØ5(7)	O$BMØ5(7)	C$MØ5(4)
MPC line printer		O$ALØ6(4)			
PRIMOS file system compressed ASCII/Binary	I$ADØ7(1)	O$ADØ7(1)	I$BDØ7(1)	O$BDØ7(1)	SEARCH(1)
PRIMOS file system uncompressed ASCII/Binary	I$ADØ7(1)	O$ADØ8(1)	I$BDØ7(1)	O$BDØ7(1)	SEARCH(1)
Serial card reader	I$ACØ9(3)				
7-track magnetic tape ASCII/Binary	I$AM1Ø(4)	O$AM1Ø(4)	I$BM1Ø(7)	O$BM1Ø(7)	C$M1Ø(4)
7-track magnetic tape BCD	I$AM11(7)	O$AM11(7)			C$M11(7)
Input command stream	I$AA12(1)				
9-track magnetic EBCDIC	I$AM13(7)	O$AM13(7)			C$M13(7)
Versatec/Gould printer/plotter		O$AL14(3)			
MPC card processor	I$AC15(3)	O$AC15(3)			

Numbers in parentheses refer to notes in the text.

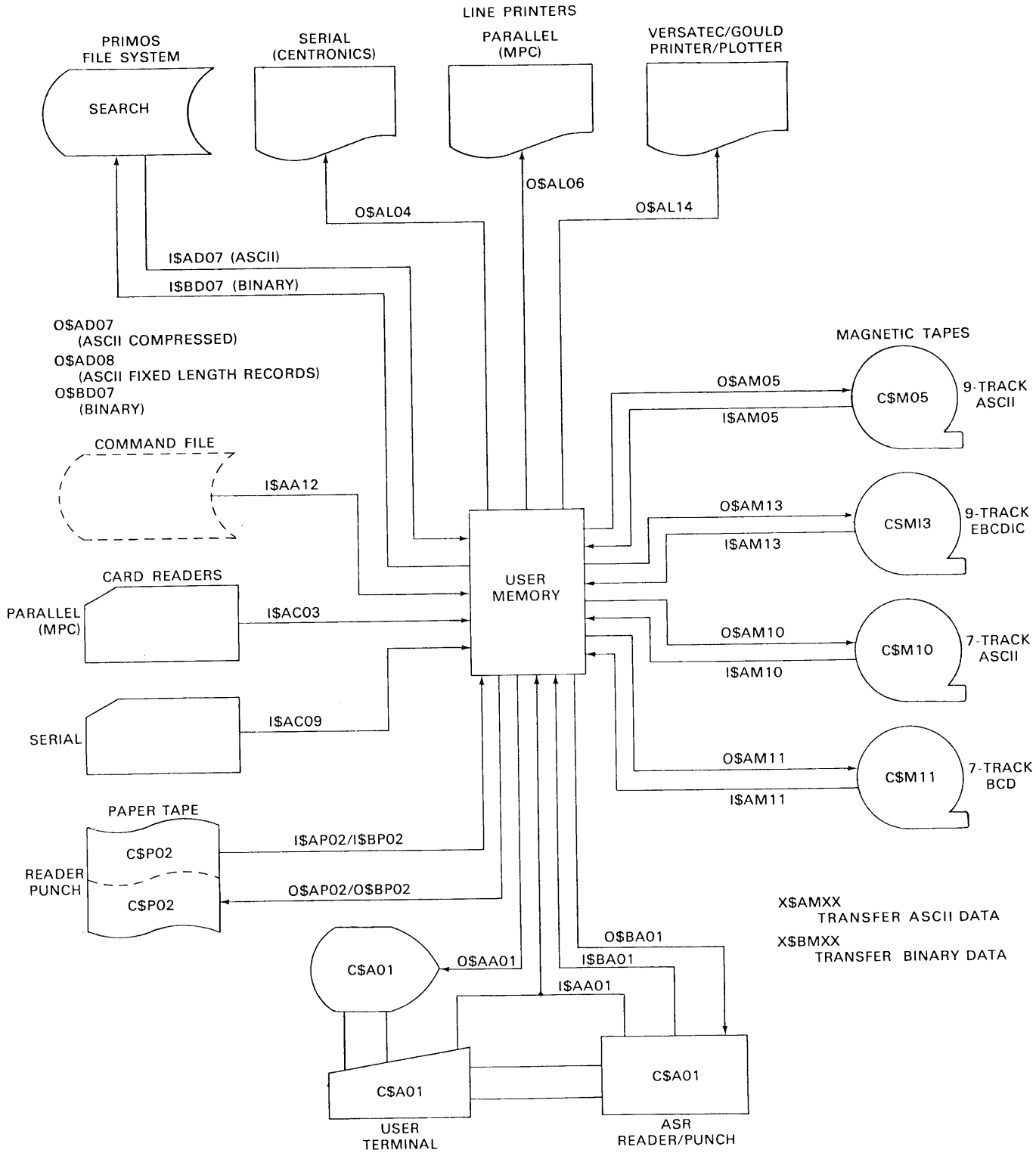


Figure 15-1. Transfer of Data to and from High-Speed User Memory

7. Available in R-mode and V-mode. V-mode is listed in FLCONV but not in CONIOC. R-mode is not in CONIOC or FULCON. In V-mode, if the non-shared library (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers unless FLCONV is assembled and loaded before the library is loaded.

   In R-mode, the appropriate table must be modified, assembled and loaded before the library is loaded.

   In both modes, the routine may be called specifically without any special procedures.

DATA FORMATS

All first and second level device drivers are uniform in the internal representation of data. All ASCII data, for example, has the same internal format regardless of the physical device.

ASCII Data

Data associated with logical I/O functions RDASC (Read ASCII) and WRASC (Write ASCII) are represented internally as an ASCII string in card image format. This string is of length N words with each word containing ASCII coded characters (N is defined in the calling sequence to the driver).

Notes

1. The "new-line" ('212) must not be used as data because it is the end-of-record indicator.

2. ASCII drivers should only be used to transfer printable ASCII characters.

Binary Data

Binary data is transferred using RDBIN and WRBIN. The external format varies considerably from device to device, but the internal format remains the same. Binary data can consist of anything and is not interpreted by the driver in any way.

The parameter buffer (buffer address) in a call to RDBIN (Read Binary) or WRBIN (Write Binary) defines the first word of the binary data. The word count on output must be defined by the user.

SUBROUTINES FOR DEVICE-INDEPENDENT DRIVERS

The device-independent drivers all have the same arguments. The arguments are:

logical device	The logical device to or from which data is to be moved. (See Table 15-1)
buffer	A buffer to or from which data is moved.
count	The number of words to be transferred.
altrtn	An integer variable assigned the value of a label in the user's FORTRAN program to be used as an alternate return in case of end-of-file or other error (Ø if no alternate return wanted).

Use of altrtn

If altrtn is omitted, the name of the device dependent driver and an error message is typed, then control returns to PRIMOS. If altrtn is not omitted, user code at the specified label must call GETERR(ERRVEC, 2) to pick up the error code. If the error code is the two ASCII characters 'IE', the error was end-fo-file. Other possible error messages and codes are given in Appendix G. If the user wishes not to handle a particular error type, he may CALL PRERR to print the error message that would have been typed if altrtn were omitted, then CALL EXIT to return control to PRIMOS. The FORTRAN statements of the form READ(nam,num,ERR=,END=) handle end-fo-file for END= and format errors for ERR=. Other errors are not handled.

Using altrtn of Ø will not work in 64V mode - the argument must be omitted!

▶ WRASC

The contents of buffer are moved from memory to the output device. The format of the data on the output medium is device-specific. Memory is assumed to consist of ASCII, two characters per word.

CALL WRASC (logical-device,buffer,count,altrtn)

▶ RDASC

One record is brought into memory. Buffer is always filled with count ASCII characters, two per word. If the record was longer than count words, buffer contains the first count words in the record and the next successive read will give the first count words of the next record, not the remaining words of the long record. If the record is less than count words, the remainder of the buffer will be blank-filled.

CALL RDASC (logical-device,buffer,count,altrtn)

▶ WRBIN

The number of words specified by count are written from buffer to the specific output device. The format of the data is device-dependent.

CALL WRBIN (logical-device,buffer,count,altrtn)

▶ RDBIN

A record is read into memory. Count is the maximum number of words which will be read into buffer. If the record is less than count long, then count will be set to the number of words actually read. If the record is longer than count, only the first count words will be read.

CALL RDBIN (logical-device,buffer,count,altrtn)

▶ CONTRL

Certain non-data transfer functions, such as opening a PRIMOS file for reading, are provided by use of the CONTRL subroutine. Functions not applicable to a particular device are ignored; therefore, functions can be requested in a device-independent way. See Table 15-2 for operation effects.

CALL CONTRL (key, name, logical-unit, altrtn)

key	1 open for reading 2 open for writing 3 open for read/write 4 close 5 delete file 6 move forward 1 file mark (MT only) 7 rewind to beginning of file 8 select device and read status (MT only). Status is returned in the A-Register. -1 write file mark (MT only) -2 backspace 1 record (MT only) -3 backspace 1 file mark (MT only) -4 rewind to beginning of tape (MT only)
name	Filename (Ø if none).
logical-device	The logical device to be controlled.
altrtn	An integer variable assigned the value of a label to be used as an alternate return in case the operation fails. (omit if no alternate return wanted).

Note

For calls to CONTRL that are directed to the disk files, key may have many other values. For disk files, CONTRL calls SEARCH with the same arguments. Keys other than 1-4 are not device-independent.

Table 15-2. List of Keys and Operating Effects for CONTRL

Key	Terminal C$AØ1	Paper Tape Read/Punch C$PØ2	Magtape C$MXX	Disk SEARCH
1	a	a	a	a
2	q	q	b	b
3	q	q	c	c
4	r	r	d	p
5	--	--	h	e
6	q	q	i	z
7	s	s	n	f
8	--	--	k	g
-1	--	--	l	z
-2	--	--	m	z
-3	--	--	n	z
-4	--	--	o	z

a open for read
b open for write
c open to read and write
d rewind and close file
e delete file
f position to beginning of file
g truncate file
h move forward one record
i move forward 1 file mark
k select device and read status
l write file mark
m backspace one record
n backspace one file mark
o rewind to BOT (beginning of tape)
p close file
q turn on punch and punch leader
r if device was open for output, punch trailer and turn off paper-tape punch and reader
s halts allowing operator to rewind tape type 'START' to continue
z abort ("Bad Key" Error)

Keys other than 1 through 4 are not device independent.

SECTION 16

DEVICE DEPENDENT DRIVERS

This set of device-dependent subroutines provides a consistent calling sequence for various non-data-transfer functions of several devices. Arguments and functions not applicable to a particular device are ignored.

SUBROUTINE CALLING SEQUENCE

The calling sequence contains all the information needed by the disk control subroutine (SEARCH), and many of the parameters are ignored by the non-disk control subroutines.

CALL xxxxx (key, name, physical-unit, [altrtn])

key	Points to a value that defines the desired function (see Table 15-2 in Section 15).
name	Points to a 1-6 character file name(or points to zero if no file name applicable). Where name is not significant, it is usually specified as Ø.
physical -unit	Points to the physical device sub-unit number. If the device has only one unit, its sub-unit number is 1. If it is a multiple unit device (cassette, mag tape, disk), sub-units 1-8 may be specified (on disk, a sub-unit is actually processed as file 1-8).
altrtn	Specifies the transfer location if an error condition is detected. This parameter is optional, and if not present, error returns will be made through the normal return (with the A register set to non-zero).

Subroutine Name ('xxxxxx') Definitions:

xxxxxx =	C$AØ1	user terminal
	C$PØ2	paper-tape reader or punch
	C$MØ5	9-track magnetic tape
	C$M1Ø	7-track magnetic tape ASCII drivers read ASCII data format (units Ø-7)
	C$M11	7-track magnetic tape ASCII drivers read BCD data format (units Ø-7)
	SEARCH	PRIMOS disk files (units 1-126)

SECTION 17

DISK SUBROUTINES

This section defines the subroutines for Disk I/O operations.

SUBROUTINE DESCRIPTION

▶ O$AD07

O$AD07 writes ASCII from buffer onto a disk file open on unit.

Information is written on the disk in compressed ASCII format. Multiple blank characters are replaced with the character DC1 (221 octal) followed by a character count. Trailing blanks are removed and the end of record indicated by the new line character, or new line followed by null.

CALL O$AD07 (logical-unit,buffer,count,altrtn)

▶ I$AD07

I$AD07 reads information from the disk open on unit, in compressed ASCII format.

CALL I$AD07 (logical-unit,buffer,count,altrtn)

▶ O$BD07

O$BD07 writes binary information to the file open on unit.

CALL O$BD07 (logical-unit,buffer,count,altrtn)

▶ I$BD07

I$BD07 reads binary information from the file open on unit.

CALL I$BD07 (logical-unit,buffer,count,altrtn)

▶ O$AD08

O$AD08 writes ASCII from buffer onto disk upon file open on unit. Information is written on the disk in fixed length records. Each record consists of count words followed by a word containing NL and NULL (105000 octal). This driver is not in the standard CONIOC supplied by Prime. It is useful in conjunction with POSFIL for those users interested in using direct access files.

CALL O$AD08 (unit,buffer,count,altrtn)

SECTION 18

USER TERMINAL SUBROUTINES

This section defines subroutines used to transfer data to and from a user terminal or Reader/Punch (ASR). Subroutines which have special options or error handling features are described in detail.

CALLING SEQUENCE

CALL {O|I}$ {A|B}zyy (sub-unit,buffer,count,altrtn)

I	Input
O	Output
A	ASCII
B	Binary
z	One-letter mnemonic giving general device class.
yy	Two digit device types.
sub-unit	Specifies unit for multi-unit device types. This parameter is ignored for single-unit device types.
buffer	Memory buffer for data.
count	Word count for transfer. Details are the same as for RDASC, RDBIN, WRASC, and WRBIN.

The device-dependent IOCS drivers are shown in Table 15-1.

KEYBOARD TERMINALS AND PAPER TAPE SUBROUTINES

User Terminal or ASR Punch	O$AAØ1	Outputs ASCII to the user terminal or ASR punch. Calls the low level driver TNOU. Errors: none.
Keyboard or ASR Reader	I$AAØ1	Inputs ASCII from user terminal or ASR reader. The kill and erase characters (question mark and quote mark) may modify the input line, as with the PRIMOS III command line. The characters NUL, DEL, DLE, DC2, DC3, and DC4 are ignored. The character ETX ('2Ø3), indicates end of file and is used in reading tapes through the user terminal. Note that I$AAØ1 is not the entry for the user terminal in the Prime-supplied CONIOC. Put I$AAØ1 in the table to read paper tapes with user programs. The editor should be used to read in the tape, then the user may read the file from the disk. Errors: none
HS Paper Tape Reader	I$APØ2	Inputs ASCII from the high-speed paper-tape reader. The kill and erase characters (question mark and quote) modify the input similar to PRIMOS III command line. NUL, DEL, DLE, DC2, DC3, and DC4 are ignored. The character ETX ('2Ø3) indicates end of file. Calls: PlIN, ERRSET Error Message: I$APØ1 EOF (IE)
HS Paper Tape Punch	O$BPØ2	Outputs binary data to the high-speed paper-tape punch. The format of the paper tape can be found in a listing of the driver.

RAW DATA MOVERS

HS Paper Tape Reader	PlIB	Input one character from the high-speed paper tape reader to the A Register. (This routine also available in in V-mode).

HS Paper Tape Punch	P10B	Output one character to the high-speed paper tape punch from the A Register. (This routine also available in V-mode).
HS Paper Tape Punch	P10U	Output one character to the high-speed paper tape punch. Zero the high order bit before punching. No special action is taken on carriage returns or line feeds. (This routine also available in V-mode.) CALL P10U (char)
ASR Reader	P1IN	Input one character from paper tape, set high order bit, ignore line feeds, send a line feed when carriage return is read.
User Terminal	TNOU	Outputs count characters to the user terminal followed by the LINE FEED, CARRIAGE RETURN. Buffer is expected to contain 2 characters per word. CALL TNOU (buffer, count)
User Terminal	TNOUA	Outputs Count characters to the user terminal. CALL TNOUA (buffer, count)
User Terminal	TOVFD$	Outputs the 16-bit integer num, without any spaces, to the terminal e.g. "123" or "-17". CALL TOVFD$ (num)
Keyboard to A Register	T1IB	Reads one character from the user terminal into the A Register.
A Register to User Terminal	T1OB	Writes one character from the A Register to the user terminal.

User Terminal to Memory	TlIN	TlIN reads one character from the user terminal. If a .CR. (CARRIAGE RETURN) is read, .NL. (NEW LINE) is output and char is set to .NL. If an .NL. is read, a .CR. is output and char is set to .NL. If .XOF. is read, carriage return and new line are expected to follow. TlIN ignores the .XOF., reads the .CR. and .LF., then sets char to .NL. The .XOF. characters are expected on paper tapes to be read on the user terminal paper-tape reader. CALL TlIN (char)
To User Terminal	TlOU	Outputs char to the user terminal. If char is .NL., the characters .CR. and .NL. are output to the user terminal. CALL TlOU (char)

For all numeric input routines, the number may be preceeded by a "-" to indicate that it is negative; but must not be a "+". Numbers may be terminated by a RETURN or a space.

User Terminal decimal Input	TIDEC	Inputs decimal number. CALL TIDEC (variable)
User Terminal Octal Input	TIOCT	Inputs an octal number. CALL TIOCT (variable)
User Terminal Hexadecimal Input	TIHEX	Inputs a hexadecimal number. CALL TIHEX (variable)
Memory to User Terminal (decimal output)	TODEC	Outputs a six-character signed decimal number CALL TODEC (variable)
User Terminal Hex Output	TOOCT	Outputs a six-character unsigned Octal number. CALL TOOCT (variable)

Carriage Return/ Line feed to Hexadecimal data format	TOHEX	Outputs a four-character unsigned hexadecimal number.

CALL TOHEX (variable)

Memory to User Terminal with carriage return	TONL	Outputs carriage return and line feed

CALL TONL

A "?" will be typed if number is improper and more input will then be accepted. A NULL input (space or return) will return a Ø.

SECTION 19

PERIPHERAL DEVICES

This section defines subroutines that control line printers, printers/plotters, card readers and magnetic tapes. These subroutines are used for both formatted and raw data.

LINE PRINTER SUBROUTINES

IOCS contains subroutines to control three types of line printers. They are: O$AL04 to print on a Centronics Line Printer connected to the SOC; O$AL06 to print on a parallel interface line printer connected to the MPC Line Printer Controller; and O$AL14 to print on a Versatec Printer/Plotter connected to a Versatec-SOC Controller. O$AL14 also prints on a Gould Printer/Plotter connected to a Gould-SOC Controller. All three subroutines have the same action on the appropriate device; therefore only one description is given. (xx below is 04, 06, or 14).

Subroutine Calling Sequence

```
CALL O$ALxx (physical-unit,buffer,count,altrtn)
```

physical-unit	Line printer unit number. 0 = PR0, first controller 1 = PR1, first controller 2 = PR2, second controller 3 = PR3, second controller
buffer	The name of the buffer where the text to be printed resides. Print text is placed in the buffer, two characters per word.
count	The number of 16-bit words of data to be printed.
altrtn	Never taken and is an optional calling sequence parameter.

Printer Control

The action taken by O$ALxx depends on the data in the buffer, and the current vertical control mode. Certain characters within the data control the manner in which the data is printed. These characters (codes) are described in the following paragraphs.

Vertical Control Modes

O$ALxx has three vertical control modes:

- FORTRAN forms control
- header line and paginate control
- no control

O$ALxx checks the first character in the data buffer for an ASCII .SOM. character (ØØ1). A .SOM. character signifies a change in the control mode. If the first character in the buffer is not an .SOM., the line is printed according to the current control mode. The default mode is FORTRAN forms control.

FORTRAN Forms Control Mode

The FORTRAN forms control mode corresponds to ANSI FORTRAN forms control conventions. The first character in the buffer is not printed; instead, it is used for forms control. The character interpretations are as follows:

Character	Interpretation
Ø	Skip a line.
1	Eject to top of next page.
+	Overprint last line.
-	Skip two lines.
Any character other than Ø, 1, +, -	No action.

Header Line and Paginate Control Mode

In Header Line and Paginate Mode, O$ALxx causes a header line to be printed, followed by three blank lines followed by 38 text lines. The header line consists of up to 43 characters followed by a page count that is generated by O$ALXX when printing in this mode.

No Control Mode

In No Control Mode, no actions are taken by O$ALxx. A line containing an ASCII form-feed (FF, :214) character causes the line preceding it to print, followed by a page eject. Carriage return (CR, :215) will cause the line preceding it to print with no spacing. Line feed (LF, :212) will cause the line preceding it to print followed by a line spacing operation. Any characters following a CR, LF, or FF are ignored.

Change of Mode Commands

Any data buffer beginning with a .SOM. character causes O$ALXX to take some action to change control mode. The control mode change is determined by the character following the .SOM. The character interpretations are:

Character	Interpretation
ØØØ	Enter No Control Mode.
ØØ1	Enter FORTRAN Control Mode.
Ø36	New Header Line - DO NOT reset page count.
Ø37	Enter new page size specified by the 16-bit number contained in the next computer word.
All other	Enter Header Control Mode characters.

Early Buffer Termination

A LINE FEED (LF, :212) character terminates the print line in the buffer, regardless of the count parameter.

Errors: none

Load information: O$AL04 calls no other subroutines. O$AL06 calls T$LMPC and O$ALI4 calls T$VG.

▶ T$LMPC

The T$LMPC routine is the raw data mover that moves information from the user to one line on the MPC line printer.

T$LMPC is called by the IOCS line printer driver O$AL06. The user normally prints lines under program control using either FORTRAN WRITE statement or a call O$AL06. However, it is possible to call T$LMPC directly.

CALL T$LMPC (logical-unit,buffer,count,instr,status)

logical-unit	Line Printer unit (currently ignored).
buffer	A pointer to a buffer to hold information to be printed on the line printer. Information is expected to be packed two characters per word.
count	Number of words to print on the current line.
instr	The instruction required to be sent to the line printer. Valid instructions are:

Instruction (Octal)	Meaning
100000	Read status
40000	Print a line
20012	Skip a line
20014	Skip to top of page
20100-20113	Skip to tape channel 0-11
20120-20137	Skip from 1 to 15 lines

status — A three-word vector that contains device code, status of printer, and a space. Possible printer status is as follows:

Octal Value	Condition
200	ON-LINE
100	Not Busy

Under PRIMOS, line printer output is buffered. If T$LMPC is called and the buffer is full, the user is placed in output-wait state. Later, when the buffer is no longer full, the user is rescheduled, and the T$LMPC call is retried. The user may issue a status request call to check if the buffer is full. If the buffer is full, then the not-busy status is reset. Using this feature, a user program may check that the buffer is not full, then output on line, or do another computation if the buffer is full.

Under PRIMOS II, output is not buffered, and control does not return to the user until printing is complete.

▶ SPOOL$

A user program can insert a file into the spool directory by calling the SPOOL$ subroutine from the applications program. This subroutine SPOOL$ is in the SPOOL$ library (R-mode) and VSPOO$ library (V-mode).

CALL SPOOL$ (key, name, namlen, info, buffer, buflen, code)

key — 1 copy named file into queue.
2 open file on unit info(2) for writing.

name — File to be copied (key=1).
Name to appear on banner (key=2).

namlen — Length of name, in characters (1-32).

info — Information array, 12 elements, as follows:

1 — temp file unit 1 (may range from 1-126, Rev. 17 and above)

2 — temp file unit 2 (may range from 1-126, Rev. 17 and above)

3 — print option word (see below)

4-6 form type (6 ASCII characters)

7 plot raster scan size (plot only) this represents #words/raster scan

8-10 spool filename (returned)

11 deferred print time (valid only if defer bit specified in option word)

12 file size, returned if key 1.

buffer Scratch buffer - this is used to set up control info and to copy the file to the spool queue (key=1) - it must be at least 40 words long. Copy time is inversely proportional to buffer size. Nominal size is between 300-2000 words.

buflen Length of buffer.

code Return code (non-zero if file system error).

Word 3 of the information array (print option word) is defined as follows:

Bit	Meaning
1	FORTRAN format control (col 1 contains carriage ctl info)
2	Expand compressed listing
3	Generate line #'s at left margin
4	Suppress header page
5	Don't eject page when done
6	No format control
7	Plot file - info (7) must be specified
8	Defer printing to specified time-info(11) must be valid
9	Print on local printer only

PRINTER/PLOTTERS

The Printer/Plotter subroutines are used to drive and control a Versatec and Gould Printer/Plotter.

▶ T$VG

T$VG exists in two versions. One version interfaces with a Versatec printer/plotter and the other interfaces with a Gould printer/plotter.

This version of T$VG moves raw data from a buffer and prints the data on the Versatec printer, connected to the Prime computer via a controller designed for use with the Versatec printer/plotter.

CALL T$VG (unit,LOC(buffer),nwds,instruction,status)

unit	Currently always Ø, since the controller supports only one device.
LOC(buffer)	Address of user's buffer.
nwds	The number of words in the buffer, currently the maximum is 18Ø.
instruction	A number from Ø to 1Ø that specifies an action that the device is to take. These instructions are described in detail in the following paragraphs.
status	A two-word status array. Device status is returned to status(2). status is returned only on a status request instruction.

The interpretation of the bits that are set in status(2) are as follows:

Bit	Meaning
1	Always zero.
2	If=1, then paper is low.
3	If=Ø, then printer/plotter is READY. If=1, printer/plotter is NOT READY.

4	If=Ø, printer/plotter is on line otherwise, printer/plotter off line.
5-16	Always zero.

Printer/Plotter Instructions

Instructions to the printer/plotter are specified in the instruction field of the calling sequence. They are a number 1 to 1Ø interpreted as follows:

Ø Return printer/plotter status in status(2). The contents of the status vector, status, are described in the calling sequence description. Under PRIMOS III, T$VG waits until the output buffer is empty before returning status. Therefore, status requests should be used sparingly.

1 End-of-transmission. This instruction initiates a print cycle and a paper advance. If the paper on the printer/plotter is installed in roll form, this roll is advanced eight inches; if the paper is fanfolded, it is spaced to the top of the next form.

2 Reset. The reset instruction clears the buffer and initializes all logic in the printer/plotter.

3 Form feed. The form feed initiates a print cycle and a paper advance.

If the paper on the printer/plotter is installed in roll form, the paper is advanced 2-1/2 inches; If the paper is fanfolded it is advanced to the top of the next form.

4 Clear buffer.

5 Reserved.

6 Print the contents of buffer (Print Mode).

7 Make a Plot, using the contents of buffer (Plot Mode).

8 Simultaneous print/plot PRINT (SPP Mode).

9 Simultaneous print/plot PLOT (SPP Mode).

1Ø Return status of output queue in status(2). If there is no room for the number of words specified by the parameter nwds, set status(2) to zero. If there is room for the number of words specified by nwds, set status(2) to a non zero value.

Print Mode: The Versatec Printer/Plotter may be operated as if it were a line printer. The printer/plotter accepts 6- or 8-bit ASCII code. Control commands are transmitted by using the instructions described for the calling sequence or by transmitting the following ASCII control codes:

ASCII Code (Octal)	Meaning
004	End of transmission.
014	Form Feed.
012	Line Feed. The transmission of a Line Feed code causes a print cycle and a paper advance of one line, except when the 012 code follows either the printing of a full buffer or a Carriage Return (015).
015	Carriage Return. A Carriage Return causes a print cycle and a paper advance of one line, provided the buffer has at least one character entered and provided the buffer is not full.

When the 8-bit (128-character) ASCII character set is used, there are no ASCII control codes.

Plot Mode: The printer/plotter performs plot operations that are standard to all printer/plotter devices connected via the controller to the Prime computer. Plot data consists of 8-bit, binary, unweighted bytes. Each dot that is plotted at the printer/plotter corresponds to a single bit in the buffer. If bit is 1, a black dot is plotted at the point corresponding to the bit position in the buffer. Bit 1 of a memory word (2 bytes) is the most significant (i.e., leftmost) bit, and Bit 16 of memory word is the least significant (i.e., rightmost) bit.

Simultaneous Print/Plot (SPP) Mode: SPP mode operation permits direct overlay of character data which is generated by an internal matrix character generator, with plotting data, which is generated on a bit-to-dot correspondence. The SPP mode is an optional feature on some printer/plotters. The SPP process makes use of both a print buffer and a plot buffer, both specified in calls to T$VG. For example, using the Printer/Plotter Model 1100A in SPP mode, the SPP operation consists of first, placing up to 132 ASCII characters in the PRINT buffer (Instruction= 8); and then placing 128 bytes of plot data in the buffer (Instruction=9) ten times. When the plot data is transmitted to the printer/plotter, the plot buffer is scanned, and a single row of dots, corresponding to the binary content of the plot buffer, is printed. During the scanning process, the print buffer is also scanned. The corresponding dots of each print character are OR'd with the plot buffer output; thus an overlay is formed consisting of the

printed and plotted data. Since the vertical height of an ASCII character for the Model 1100A Printer/Plotter is ten raster scans, the user must make ten calls to plot data before the print buffer is completely printed and ready for new data. Table 19-1 shows the number of raster scans per print line for the various models of Versatec Printer/Plotter optionally available with Prime computer configurations.

Caution

For SPP mode, do not attempt to transfer more than the maximum number of characters to the print buffer.

SPP mode requires a series of calls to the T$VG driver. For instance, in the example given, each print instruction was followed by ten plot instructions. Do not interrupt such a sequence with other instructions, because printer/plotter output will be incorrect.

Table 19-1. Maximum Buffer Length for Versatec Printer/Plotters.

	PLOT			PRINT No. Scans/Print Lines	
Model	Bits	Bytes	Chars.	64 Chars.	96 or 128 Chars.
220a	560	70	80(70 in spp)	8	10
1100a	1024	128	132	10	12
1600a	1600	200	100	20	20
2000a	1856	232	232	10	12
2160a	2880	360	180	20	20

▶ T$VG

This version of T$VG moves raw data from a buffer and prints the data on the Gould Printer/Plotter, connected to the Prime Computer via a controller designed for use with the Gould Printer/Plotter.

CALL T$VG (unit,LOC(buffer),words,inst,status)

unit 0 - controller supports one device.

LOC (buffer) Address of user's buffer.

words Number of words for output (maximum is 180).

inst Ø - 1Ø (described below).

inst	Meaning
Ø	Return device status in status(2). Status is returned only on a status request instruction. Under PRIMOS III, T$VG waits until buffer is empty, then returns status. Therefore, status requests should be done sparingly.

status(1) always Ø

status(2) bits:

bits	
1	1 for Gould
2	paper low if = 1
3	Device ready if = Ø
4	Device on-line if = Ø
5-15	Always zero
16	Device was deselected during last operation. This indicates an error condition only in high speed graphics mode.

1	End-of transmission. This initiates paper advance of 2.56 inches.
2	Not used.
3	Form feed. This initiates a paper advance 1.28 inches of roll mode or to top of next page if fan-fold mode.
4	Not used.
5	Cut paper.
6	Print words words from buffer.
7	Plot words words from buffer.
8	Set high-speed plot mode and plot words words (PRIMOS II only).
9	Reset high-speed plot mode and plot words words (PRIMOS II only).

10 Return status of output queue in status(2). Set status(2) to zero if no room for N words. Otherwise set it equal to zero.

status Two-word status array.

Note

For instructions 6-7, the driver automatically initiates the necessary write cycles to print or plot the outputted data.

Print Mode: The Gould accepts ASCII 7 or 8 level code. Control commands may be transmitted by using the above instructions. A bad code prints as solid black square.

Plot Mode: Plot operations are applicable to all matrix plotters and printer/plotters. Plot data consists of 8-bit, binary, unweighted bytes. For the number of bits per a complete raster scan, see the maximum buffer length list below. Each dot corresponds to a single bit in the buffer. If a bit is a '1', a black dot is plotted at the point corresponding to the bit position in the buffer. Memory bit 01 is the MSB, bit 16 is the LSB.

High-Speed Plot Mode: High-Speed Plot Mode is available only under PRIMOS II. A user must call T$VG to set high- speed plot mode and plot a line. He then must call T$VG every 90 milliseconds or less to plot a raster line (inst = 8) or gaps will appear in his plot. The last two lines of the plot must be generated with calls to T$VG with instructions of 9 and 7, respectively. These two special calls are required so the paper will decelerate and stop following the last lines. A user may check to see if he is calling T$VG to plot often enough by calling T$VG to get status after every plot call. Bit 16 of status(2) will be on if the user has failed to give the previous two plot requests closely enough in time. Bit 16 should be ignored except for checking in high-speed plot modes.

Maximum Buffer Length

Model	Bits	Plot Bytes	Print Characters
4821	600	75	85
4822	800	100	114
5000	1024	128	132
5100	2048	256	264

▶ O$AL14

O$AL14 provides the IOCS interface to the Versatec Printer.

CALL O$AL14 (buffer,count,altrtn)

buffer Buffer to/from which data are moved.

count Number of words to be transferred.

altrtn Never taken and is an optional calling sequence.

The action taken by O$AL14 depends upon the data in the buffer and the current vertical control mode.

O$AL14 has three vertical control modes:

1. FORTRAN forms control
2. Header line and paginate control
3. No control

The default mode is FORTRAN forms control. O$AL14 checks the first character in the data buffer for an ASCII .SOM. (Ø18). An .SOM. character signifies a change in the control mode. If not an .SOM. the line is printed according to the current control mode. Mode descriptions follow:

FORTRAN Forms Control: This mode of O$AL14 honors ANSI FORTRAN forms control conventions. The first character in a buffer is never printed but is used for forms control. The character interpretations are:

Ø Skip 1 line

1 Eject to top of next page

\+ Print over last line (Not currently honored)

Other No action

Header line and paginate: In this mode O$AL14 permits a header line followed by three blank lines, followed by 56 text lines. The header line is 42 characters followed by a page count which is kept automatically by O$AL14 when in this mode.

No Control: In this mode no automatic actions are taken except that any line containing a form-feed character will cause a page eject with no further action.

Any data buffer beginning with an .SOM. will cause an internal change by O$AL14. The change is determined by the character following the .SOM.. The character interpretations follow:

ØØØ	Enter non-control mode
ØØ1	Enter FORTRAN control mode
Ø36	New header line but do not reset page count
All others	Enter header control mode

When entering header control mode the characters following the .SOM. are stored internally in O$AL14 for use as the header line.

All change of mode commands cause a page eject before any further action.

CARD PROCESSING SUBROUTINES

CARD READER subroutines drive and control serial and parallel interface type card readers.

▶ I$ACØ3

Reads ASCII input from the parallel interface Card Reader.

CALL I$ACØ3 (unit,buffer,word-count,altrtn)

unit	Logical device to or from which data is to be moved. Ø = CRØ, first controller 1 = CR1, second controller
buffer	Buffer which receives data from card reader.
word count	Number of words to be transferred.

altrtn — Alternate return in case of end of file or other error.

Card Format: Cards are expected to be in Ø29 format. 'Ø26' cards may be read by preceding the deck by a card containing '$6' in columns 1 and 2. The conversion done for 'Ø26' cards is shown below.

Card Code (Ø26 Symbol)	Converted to (Character)
#	=
%	(
<	)
@	'
&	+

The driver can be switched back to 'Ø29' format by '$9' in columns 1 and 2.

▶ I$ACØ9

The subroutine I$ACØ9 reads ASCII input from a serial interface card reader.

CALL I$ACØ9 (unit,buffer-name,word-count,altrtn)

Load Information: I$ACØ9 calls F$AT to fetch the arguments.

Note

I$AC09 translates card codes to characters in memory as follows:

Card Code (Ø26 Symbol)	Converted to (Character)
#	=
%	(
<	)
+	&
&	+
@	'

Card codes read are either Ø26 or Ø29. The last card in the deck is .Q.

The ERRVEC(3) may have the following octal values. Combinations are possible.

200 on line

40 illegal ASCII

20 DMx overrun

4 hopper empty

2 motion check

1 read check

▶ I$AC15

Reads and interprets (prints) a card from a parallel interface card reader.

CALL I$AC15(unit,buffer,word-count,altrtn)

unit	Card reader unit. Ø = CRØ, first controller 1 = CR1, second controller
buffer	Buffer into which card is to be read.
word-count	Number of words to be read.
altrtn	Alternate return in case of error.

▶ T$CMPC

The T$CMPC routine is the raw data mover that moves a card of information from the MPC card reader to the user's space.

CALL T$CMPC(unit,LOC(buffer),word-count,instruction,status)

T$CMPC is called by the IOCS card reader driver I$ACØ3. The user normally reads cards under program control using either FORTRAN READ statement or a call to I$ACØ3. However, it is possible to call T$CMPC directly.

unit	Card reader number.
LOC(buffer)	A pointer to a buffer to hold a card of information read from the card reader.
word-count	The number of words to be read from the current card.
instruction	The instruction required to be sent to the card reader. Valid instructions are:

Instruction	Meaning
1ØØØØØ (octal)	= Reads status
4ØØØØ (octal)	= Read card in ASCII format
6ØØØØ (octal)	= Read card in Binary format

status	A three-word vector.

status(1) Not used.

status(2) Card reader status.

Octal Value	Condition
200	ON-LINE
40	Illegal ASCII
20	DMX overrun
4	Hopper Empty
2	Motion Check
1	Read Check

status(3) Number of words moved.

Example

```
40    DO 70 I = 1, 23
50    CALL T$CMPC (0, CARDS, 40, :40000, STATUS)
60    CALL O$....
70    CONTINUE
```

The above example reads an 80-character card of ASCII data and places the contents in CARDS.

Card Reading Operation

Under PRIMOS III and PRIMOS, card reader input is buffered. The user must insert the card deck in the card reader and give the command:

ASSIGN CRn

n = 1 for PRIMOS III, 0 or 1 for PRIMOS

About ten cards are read to fill the input buffer (this is called read-ahead and serves to buffer input). The user then starts the program that uses the card reader by calling subroutines T$CMPC, T$PMPC (at system level) or I$AC03, I$AC15 (FORTRAN library). If T$CMPC is called and the buffer is empty, the user will wait until more data is read.

The user may issue a status request call to check if the input buffer is empty. If the buffer is empty, the ON-LINE status bit (bit 9 in the status word) is reset.

Note

Under PRIMOS II, card reader input is not buffered and the card reader is never OFF-LINE.

CARD PUNCH subroutines drive and control parallel interface type card punches.

▶ O$AC03

Writes (punches) output to the parallel interface card punch.

CALL O$AC03(unit,buffer,word-count,altrtn)

unit	Card punch unit number. Ø = CRØ, first controller 1 = CR1, second controller
buffer	Buffer containing line to be punched.
word-count	Number of words to be punched.
altrtn	Alternate return in case of error.

▶ O$AC15

Writes (punches) output to the parallel interface card punch and interprets the line (prints on card).

CALL O$AC15(unit,buffer,word-count,altrtn)

unit	Card punch unit number. Ø = CRØ, first controller 1 = CR1, second controller
buffer	Buffer containing line to be punched.
word-count	Number of words to be punched.
altrtn	Alternate return in case of error.

▶ T$PMPC

T$PMPC is the raw data mover for the card punch. It is called by O$AC03, O$AC15 and I$AC15, the card punch drivers. These routines may be called by the user.

CALL T$PMPC (unit, LOC(buffer), word count, inst, status)

unit — Card punch unit.

LOC(buffer) — A pointer to a buffer that holds data to be punched. In ASCII mode, data are packed two characters per word.

In binary mode, card punches are mapped into a 16-bit word as follows:

bit	punch row
1-4	not used
5	12
6	11
7-16	0-9

word count — Number of words to punch on a card from buffer.

inst — Instruction required to be sent to card punch.

Instructions are:

Bit Set	Instruction	Meaning
1	:100000	Read status
3	:20000	Process in binary mode
4	:10000	Feed a card
5	:4000	Read a card
6	:2000	Punch a card
7	:1000	Print a card
8	:400	Stack a card

To punch a card, instruction would be an octal :12400 meaning, for example:

1. Feed a card
2. Punch a card and
3. Stack a card

status Three word status vector:

status(1) Not used
status(2) Device status

value	condition
:200	On-line
:40	Illegal code
:10	Hardware error
:4	Operator intervention required

status(3) Number of words read

MAGNETIC TAPES

The magnetic tape subroutines drive and control 7-and 9-track magnetic tape devices. The subroutine names are:

9-Track

C$M05	Control for 9-track ASCII and Binary
C$M13	Control for 9-track EBCDIC
O$AM05	Write ASCII
I$AM05	Read ASCII
O$BM05	Write binary
I$BM05	Read binary
O$AM13	Write EBCDIC
I$AM13	Read EBCDIC

7-Track

C$M10	Control for 7-track ASCII and Binary
C$M11	Control for 7-track BCD
O$AM10	Write ASCII
I$AM10	Read ASCII
O$BM10	Write binary
I$BM10	Read binary
O$AM11	Write BCD
I$AM11	Read BCD

Restrictions

Currently, PRIMOS supports record sizes up to 6K words for 9-track tapes and up to 4.5K words for 7-track tapes. Primos III does not support record sizes larger that 512 words for 9-track ASCII, EBCDIC, or binary records. There is no restriction under PRIMOS II. PRIMOS III does not support record sizes larger than 340 words for 7-track ASCII, BCD, or binary records. Under PRIMOS II, larger records may be used only if the user declares a labeled common area in his own program called MTBUF7. The common area must have an array as its first entry which is used as an expansion buffer when reading or writing 7-track magnetic tapes. The array must be 1.5 times as large as the biggest record the user intends to use. Alternately, the subroutine MTBUF7 in UFD IOCS can be modified appropriately and the FORTRAN library rebuilt.

▶ C$M05, C$M10, C$M11, C$M13

Since the subroutines are similar, they are described in groups.

CALL { C$M05 / C$M10 / C$M11 / C$M13 } (key, name, unit, altrtn)

key
-4 for Rewind to BOT (Beginning of Tape)
-3 for Backspace one file mark
-2 for Backspace one record
-1 for Write file mark
1 for Open to read
2 for Open to write
3 for Open to read/write
4 for Close (Write file mark and rewind)
5 for Move forward one record
6 for Move forward one file mark
7 for Rewind to BOF (Beginning of File)
8 for Select device and read status.

name
Not applicable (may be anything).

unit
0, 1, 2, or 3 (depending on which device is ASSIGNed).

altrtn
The alternate return. If altrtn = 0, it means that an alternate return is not desired.

These routines call T$MT and ERRSET.
Errors:

Message	Meaning	ERRVEC(1)	ERRVEC(2)
C$Mxx EOF	End-of-file	IE	1
C$Mxx EOT	End-of-tape	ID	2
C$Mxx MTNO	Magtape not operational	ID	3
C$Mxx PERR	Parity error	ID	4
C$Mxx HERR	Hardware error	ID	5
C$Mxx BADC	Bad call	ID	6

All the other subroutines have the same calling sequence. It is:

CALL subroutine (unit, buffer, n, altrtn)

unit — Unit number = Ø, 1, 2, or 3.

buffer — Buffer.

n — Number or words to be read or written. If N = Ø, then the subroutine is to write a file mark.

altrtn — Is the alternate return. If altrtn = Ø, it means that alternate returns are not desired.

These subroutines all call T$MT and ERRSET.

Errors:

Message	Meaning	ERRVEC(1)	ERRVEC(2)
subroutine EOF	End-of-file	IE	1
subroutine EOT	End-of-tape	ID	2
subroutine MTNO	Magtape not operational	ID	3
subroutine PERR	Parity error	ID	4
subroutine HERR	Hardware error	ID	5
subroutine BADC	Bad call	ID	6

Note

Parity error, PERR, occurs only after 25 parity or raw errors.

▶ T$MT

The T$MT routine is the raw data mover that moves information from magnetic tape to user address space, or from the user space to tape. T$MT also performs other tape operations, such as backspacing, forward spacing and density setting. If T$MT is called without the code argument, and an error condition is encountered, T$MT exits to the user command level, rather than the calling program. If T$MT is called with the code argument, the appropriate error code will be returned to the calling program.

CALL T$MT (unit, pba, nw, instr, statv, code)

unit Magnetic tape drive - may be either physical (Ø-7) or logical driver number. (INTEGER*2)

pba A pointer to a buffer address from which to read or write a record of information (INTEGER*4). If neither a read or write operation, pba is Ø.

nw Number of words to transfer. This number must be between Ø and 6K words (INTEGER*2). 6K words can be transferred under PRIMOS only if the buffer starts on a page boundry. Otherwise, the maximum size is reduced by the offset of the buffer from the page boundry.

instr The instruction request to the magnetic tape drivers. Valid instructions are:

Octal	Hexadecimal	Meaning
000040	0020	Rewind to BOT, 7,9-track
022100	2440	Backspace one file mark, 9-track
020100	2040	Backspace one file mark, 7-track
062100	6440	Backspace one record, 9-track
060100	6040	Backspace one record, 7-track
022220	2490	Write file mark, 9-track
020220	2090	Write file mark, 7-track
062200	6480	Forward one record, 9-track
060200	6080	Forward one record, 7-track
022200	2480	Forward one file mark, 9-track
020200	2080	Forward one file mark, 7-track
100000	8000	Select transport, 7&9-track
042220	4490	Write record, one character per word, 9-track
042620	4590	Write record, two characters per word, 9-track
042200	4480	Read record, one character per word, 9-track
042600	4580	Read record, two characters per word, 9-track
052200	5480	Read and correct record, one character per word, 9-track
052600	5580	Read and correct record, two characters per word, 9-track
040220	4090	Write binary record, one character per word, 7-track
040620	4190	Write binary record, two characters per word, 7-track
044220	4890	Write BCD record, one character per word, 7-track
044620	4990	Write BCD record, two characters per word, 7-track
040200	4080	Read binary record, one character per word, 7-track
040600	4180	Read binary record, two characters per word, 7-track
044200	4880	Read BCD record, one character per word, 7-track
044600	4980	Read BCD record, two characters per word, 7-track

Note

The following instructions are only valid with version two or three (in some cases both versions) magnetic tape controllers. Use of these instructions with older versions of the controller will cause an error message to be printed and the command will be aborted. A description of use of these commands is found later in this section.

100020	8010	Erase a three-inch gap on the tape (Version 2 and 3 controller).
100040	8020	Unload. Rewind tape and place drive offline (Version 2 and 3 controller).
100100	8040	Set density to 1600 BPI (Version 2 and 3 controller).
100120	8050	Set density to 6250 BPI (Version 3 controller).
100060	8030	Set density to 800 BPI (Version 2 controller only).
043500	4740	Read record backwards (Version 3 controller).

statv 8 word status vector containing the following:

statv(1) Status flag - 1 = operation in progress, 0 = operation finished.

statv(2) Hardware status word from magnetic tape controller. Possible values are:

Bit 01 vertical parity error
02 runaway
03 CRC error
04 LRC error
05 false gap/insufficient DMA range
06 uncorrectable error
07 raw error
08 file mark detected
09 selected transport ready
10 selected transport on-line
11 selected transport end of tape detected
12 selected transport rewinding
13 selected transport beginning of tape detected
14 tape write protected
15 DMX over-run or no formatter
16 rewind complete

statv(3) Number of words transferred (read and write operations only).

statv(4-8) Reserved for future use.

code Specifies that the appropriate error code is to be returned to the calling program. If this argument is omitted, ERRRTN is called and program will exit to user command level in case of error. The possible error codes returned are:

E$NASS Device specified in unit, not assigned.

E$IVCM Invalid command (e.g. attempt to set density on Version Ø controller).

E$DNCT Device specified in unit not connected, or no controller.

E$BNWD Invalid number of words (nw <Ø or >6144).

Magnetic tape I/O is not buffered under PRIMOS. A call to T$MT returns immediately before the operation is complete. When the magnetic tape operation is completed, the status flag in the user space is set to Ø. Therefore, a user program may loop waiting for completion and do another computation while waiting. If a user initiates another call to T$MT before the first call has completed its magnetic tape operation, the second call does not return to the user until the first magnetic tape operation has been completed.

Density Selection: It is assumed that tapes are written with one density. This assumption is enforced by only permitting changes in density at the load point. For this reason, it is not necessary, or possible, to set the density when reading a tape. When the first record is read, the density of the tape is determined. The rest of the tape will be read (or written) using that density.

For example, if the user set the density to 625Ø BPI with the ASSIGN command and read the first record of a 16ØØ BPI tape, then the rest of the tape would be read using 16ØØ BPI. If after reading that record, a record was written onto the tape (without rewinding to the load point); then that record would also be written at 16ØØ BPI. If the tape was rewound and then a record was written, the density would be switched to 625Ø BPI. Although the density setting of 625Ø BPI is remembered, it will not go into effect until a record is written at the load point.

If the user assigns a tape without specifying a density, the unit will be left at the density from the previous use. The default density (at system initialization time) is 16ØØ BPI.

Read Record Backwards: This request causes the tape to read a record while moving the tape backwards. It is sometimes possible to read a record backwards when a bad tape prevents reading the record in the forward direction. After the record is read, it will be necessary to reorganize the data. The words of the record will be in reverse order. Each word will have the bytes reversed. The bits within each byte will be in correct order.

Use of the T$MT WAIT Semaphore: Looping on the status done word uses up CPU time while the process waits for the tape operation to complete. This is not a good practice for two reasons. First, it ties up the CPU needlessly and slows down system performance in general. Second, it causes the process to waste some of its time slice without doing useful work. This will result in the process being scheduled extra time and the real time of program execution will be longer than necessary.

This problem can be solved by using a semaphore. If the process waits on a semaphore, the wait time is not counted against its time slice. Therefore, as soon as the tape operation completes, the process will be scheduled to run again to finish up its time slice.

The program T$MT contains a wait semaphore that can be used for this purpose. This semaphore is used to queue tape requests. If the process makes a tape request when the controller is busy with another operation, the process is put on the wait semaphore.

When the program wants to wait for a tape operation to complete, it can call T$MT with a request for status. Since the tape controller is already busy with the previous operation, the process will be put on the T$MT wait semaphore.

Since the status request is fast and doesn't affect the tape, it is a convenient tape operation to use to provide the semaphore wait. A scratch status vector should be used so that the status from the original call is not destroyed. Example of wait code:

```
        . . .

        INTEGER STATV(3)          /* STATUS VECTOR SET BY T$MT
        INTEGER UNIT              /* MAG TAPE DRIVE NUMBER (0-7)
        INTEGER BUF (1024)        /* OUTPUT BUFFER
        INTEGER XSTATV (3)        /* SCRATCH VECTOR FOR WAIT

        . . .
        CALL T$MT (UNIT,LOC(BUF),,:042620,STATV) /* WRITE 1024

        . . .                     /* OVERLAP EXECUTION WITH IO

C       WAIT FOR TAPE WRITE TO COMPLETE.

100     IF (STATV(1).EQ.0) GOTO 120 /* SEE IF IO IS ALREADY DONE
           CALL T$MT (UNIT,LOC(0),0,:100000,XSTATV) /* WAIT
           GOTO 100
 120    . . .
```

Error Recovery On Writing

There are many possible error recovery schemes. The two that are described here are based on different record formats. The first algorithm can be used when records contain only data. The other scheme requires that the records contain extra information for error recovery.

The following schemes are provided as alternatives to using the IOCS routines that FORTRAN uses. The error recovery provided in the IOCS routines correspond to that described for Simple Write Error Recovery.

Simple Write Error Recovery: The aim of the simple error recovery program is to get by a possible bad spot on the tape by erasing part of the tape where the error occurred and rewriting the record after that gap.

The program does not try to rewrite the record on the same spot on the tape even though repeated tries on the same spot may improve the tape enough to permit the write to succeed. The tape is considered marginal at that spot and may not be readable at a later date.

Only the version three controller (MPC-3), which supports the 6250 BPI tape drives, has an erase command. On other controllers, the tape can be erased by writing a file mark and then backspacing over the file mark. This will cause three inches of tape to be erased.

Program steps for write error recovery:

1. Check if error recovery is possible. Don't attempt error recovery if the tape drive is offline or not ready, or the tape is file protected.

2. Backspace over the record.

3. Erase a three inch gap on the tape.

   - Write a file mark.

   - Backspace a record and check that the file mark detected bit is set in the status word.

4. Attempt to re-write the record.

5. If the record was not written successfully, repeat steps 1-4 up to twenty times (a maximum of five feet of erased tape).

Write Error Recovery with Sequence Numbers: There is a drawback to the first scheme. Since the tape is bad at the spot where the error recovery is being done, it is possible for errors to occur while backspacing. For example, if the bad record has a gap in the middle of it, the program might detect two short records when backspacing. If the program has some way of identifying records, the program can be sure that it has not lost position during error recovery.

One way to do this is to include a sequence number with every record. Then when error recovery is attempted, the program backspaces two records and then reads a record. This record should contain the sequence number of the last good record before the error record.

Program steps for error recovery:

1. Check if error recovery is possible. Don't attempt error recovery if the tape drive is offline or not ready, or the tape is file protected.

2. Position the tape after the last good record.

   - Backspace two records. This will place the tape before the last good record.

   - Read a record and verify that its sequence number matches the one expected for the last good record.

   - If the 'good' record can't be read, then it is possible that the tape is not positioned correctly. Backspace several records and read those records to find the sequence number of the last good record written.

3. Erase a three inch gap on the tape.

   - Write a file mark.

   - Backspace a record and check that the file mark detected bit is set in the status word.

4. Attempt to write the record again.

5. If the record was not written successfully, repeat steps 1-4 up to twenty times, lengthening the gap each time.

Error Recovery On Reading

Error recovery when reading a tape involves repeatedly rereading the record. The problem of losing position can occur when doing error recovery. Therefore, the procedure can be improved by verifying the sequence number each time a record is read.

Program steps for read error recovery:

1. Check that error recovery is possible. Don't attempt error recovery if the tape drive is offline or not ready.

2. Backspace and reread the record eight times.

3. If unsuccessful, backspace eight records (or to the load point if less than eight records away), space forward seven records and then read the problem record. This sequence draws the tape over the tape cleaner and could dislodge a possible dirt particle.

4. Repeat steps 1-3 eight times.

Part V
Communication Controllers and Real-Time Subroutines

SECTION 20

SYNCHRONOUS AND ASYNCHRONOUS CONTROLLERS

SYNCHRONOUS CONTROLLERS

This section defines the raw data mover for the assigned SMLC line.

▶ T$SLC0

The SMLC driver is loaded in PRIMOS. A user program communicates with the driver via FORTRAN-format calls to T$SLC0. The driver communicates with the user address space via buffers in the user address space specified by the user program. The data structure, which is used by the driver, is "control block" and is provided by the user. This control block is created by the user program in the user address space. It contains pointers to the user status buffer and pointers to buffers containing a message to be transmitted or set to receive a message. A separate control block is required for each line.

CALL T$SLC0 (key,line,LOC(block),nwds)

key 1 Stop line. Only key + line required.

2 Define control block. The block is structured as in Table 20-1. It defines an area to store status information and, optionally, a message chain for reception or transmission.

3 Array block contains five words which are to be output to the controller - see Table 20-2 through 20-11 for details.

4 Array block contains a word which is to be used as the next data set control word. See Table 20-12 for details.

5 Array block contains two words which are to be used as the next receive/transmit enable words. See Table 20-13 for details.

6 The calling user process will go to sleep. It will waken at the next SMLC interrupt or after approximately 1 second. It will run with a full time slice interval. The value line is ignored, as are LOC(block) and nwds. If, however, the user process does not own any SMLC lines, the call will return immediately.

7 Return model number. Model number will be returned in block. When using this key, nwds must = 1. The possible model numbers and their associated protocols are:

Model Number (Octal)	Protocols
0	HSSMLC
5646	BISYNC and HDLC
5647	BISYNC and PACKET
5650	BISYNC and 1004/UT200/7020
5651	HDLC and 1004/UT200/7020
5652	PACKET and 1004/UT200/7020
5653	HDLC and PACKET
5654	BISYNC and GRTS

Note

Before calling T$SLC0 to configure a line (KEY=3), a call with (KEY=7) should be made to see if the Multi-line Data Link Controller (MDLC) contains the proper protocol and to determine what the line configuration should be. If an error occurs during initialization, the following error messages are printed:

No SMLCxx -(controller address)
No CONTROLLER CONFIGURED for SMLCyy (logical number)
UNDEFINED CONTROLLER ID for SMLCxx (controller address)

It is the responsibility of the caller to see that the line configuration is correct for the model of MDLC being used.

line Octal line number 0-3.

LOC(block) Address of user's block. User's block must reside entirely within one page.

nwds Number of words in block.

Timing

The user space program runs asynchronously with message transfers. A call to T$SLCØ returns immediately after executing whatever control function was required. The progress of the communication must be monitored by the user program by examination of the user space status buffer contents.

Assigning Communication Lines

The communications lines must be assigned to a user space before they can be used. The proper command is:

$$\text{ASSIGN} \quad \left\{\begin{matrix}\text{SMLCØØ}\\ \text{SMLCØ1}\\ \text{SMLCØ2}\\ \text{SMLCØ3}\end{matrix}\right\}$$

given at the user terminal. One or more lines may be assigned to a given user.

Table 20-1. Key=2 SMLC Control Block

Word 0	Last receiver/transmitter enable word sent to the HSSMLC by the driver. (This word is written into but not read by the driver.) Bit 15 = 1 : Transmitter on Bit 16 = 1 : Receiver on
Word 1	Bit 1 : Valid line enable order in bits 2-16 Bits 2-16 : Line enable order. See Table 20-4, word 0.
Word 2	Bits 1-4 : Data set status mask (DSSM) Bits 5-8 : Required data set status (RDSS) Bit 9 : Set: no data set order - ignore word 2 Bits 13-16 : Data set control order (DSCO)

Note

Issue DSCO, wait for (DS status .AND. DSSM) = RDSS, then issue line enable order

Word 3	Spare
Word 4	Pointer to top of status buffer
Word 5	Pointer to bottom + 1 of status buffer
Word 6	Pointer to next word in status buffer to receive the status information. (This word is written into but not read by the driver.)

Note

The status buffer must be completely contained in the same page as the control block.

Table 20-1. Key=2 SMLC Control Block (continued)

Word 7	Bits 1-2	: '01' there exists a continuation control block
	Bits 3-6	: Word count of next block - 8
	Bit 7	: 0
	Bits 8-16	: Offset in current 512 word page of next block

Note

The continuation block must reside in the same page as the control block from which it was continued.

Word 8	Bit 16	Set: Transmit Clear: Receive

Note

If Word 8 given (nwds > 8) then at least one DMC address pair must be given.

Words 9-10	DMC start and end address pointers.
11-12	Up to four pairs may be specified to allow
13-14	for channel chaining.
15-16	

Note

Transmit/receive buffers may reside in any page, but their starting and ending address pointers must reside in the same page.

Table 20-2. Key=3 Line Configuration Control Block (Bits 10-16)

Word 0 Bits 10 through 16, are constant for all controllers and protocols. Bits 1 through 9 for each controller follow.

Bit 10 : Enable formatter option (Bi-Sync, UT200, ICL 7020, 1004, Packet Switch depending on HSSMLC options)

Bit 11 : Enable reporting of data set changes by interrupt and status word.

Bits 12-14 : 12 13 14

- 14 — Automatic Parity Enable
- 13 — Parity Select 0 = odd, Note *
- 12 — Parity Enable

*Automatic Parity Enable appends a parity bit to the data while Parity Enable steals the most significant bit of each data byte.

Bits 15-16: 15 16

- 15 16 — Number of bits per character

If Automatic Parity is enabled with eight-bit data enabled, no parity will be generated or checked (i.e. no nine bit data formats)

Table 20-3. Key=3 Line Configuration Control Block (HSSMLC, bits 1-9).

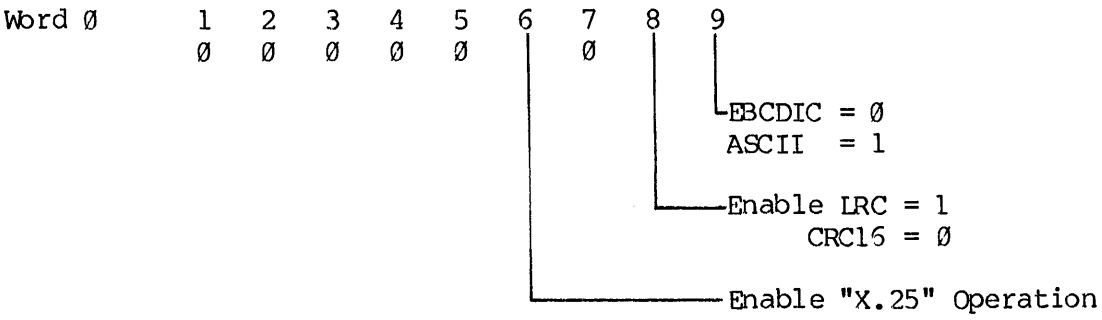
HSSMLC

Word 0 1 2 3 4 5 6 7 8 9

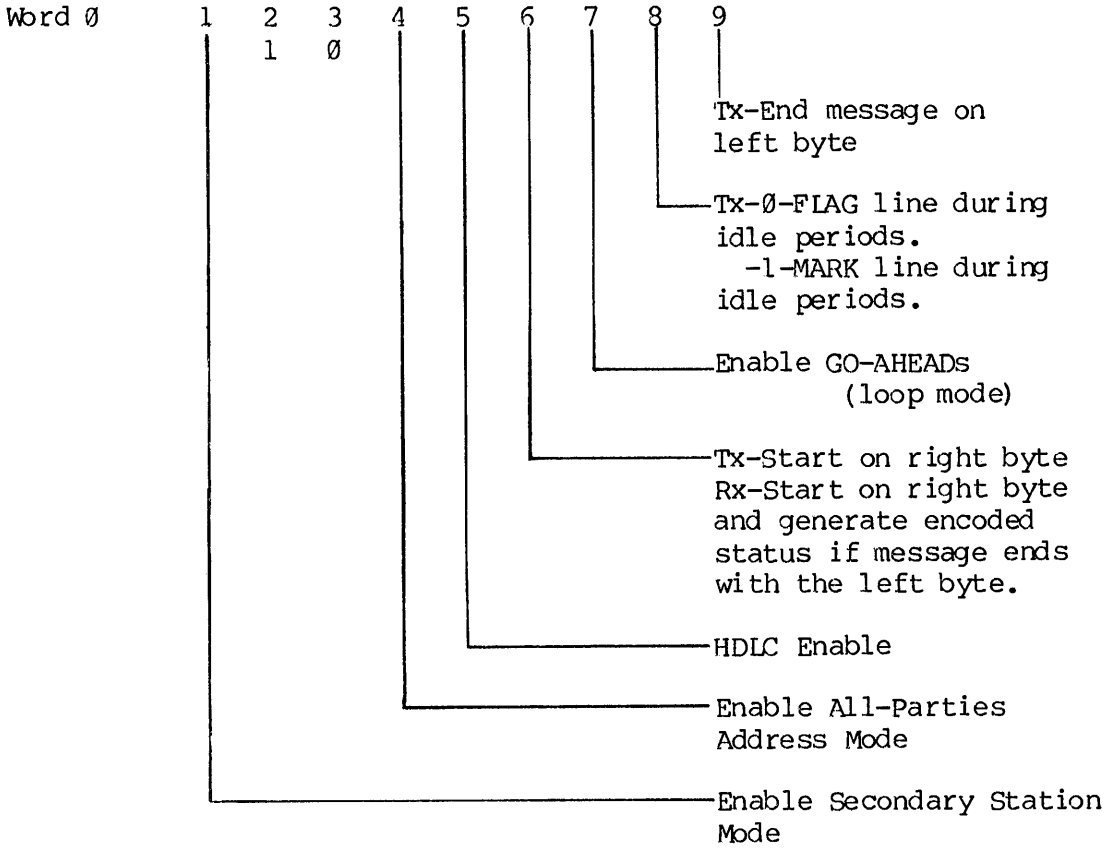
- 9 — Select formatter mode
  0 = EBCDIC
  1 = ASCII
- 8 — Select BCC
  1 = LRC (for use with ASCII mode only)
  0 = CRC-16
- 5 6 7 — Unused control bits

Table 20-4. Key = 3 Line Configuration Control Block (5646, bits 1-9).

5646
BISYNC



HDLC

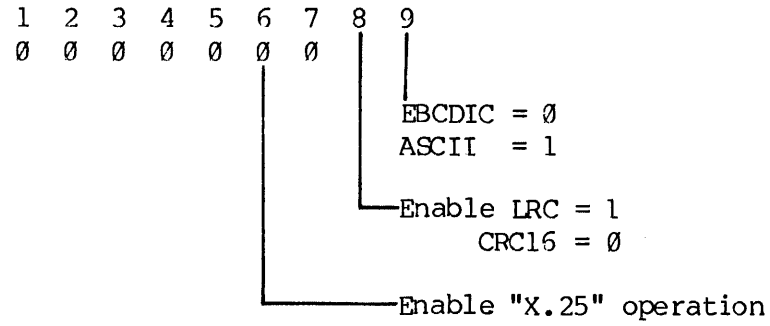


Secondary Station Mode, HDLC mode, Loop mode, and All Parties Address Mode are enabled on a line-pair basis only.

Table 20-5. Key=3 Line Configuration Control Block (5647, bits 1-9).

5647
BISYNC

Word 0



PACKET

Word 0

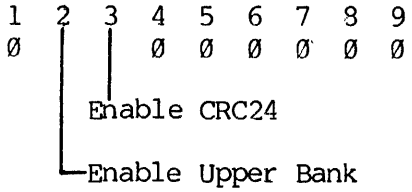
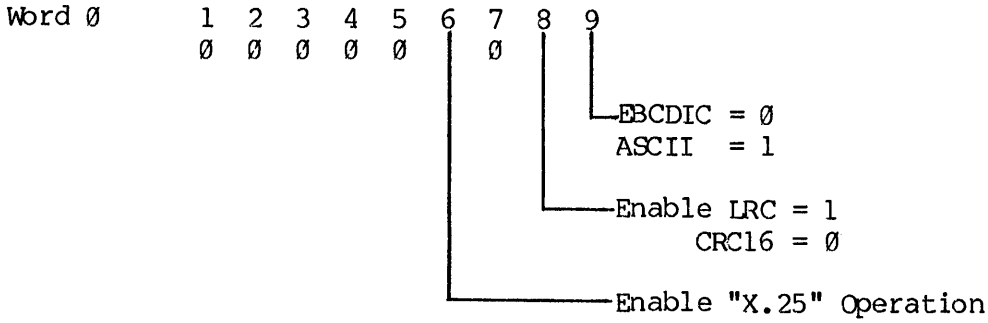


Table 20-6. Key = 3 Line Configuration Control Block (5650, bits 1-9).

5650
BISYNC



ICL7020/UT200/1004

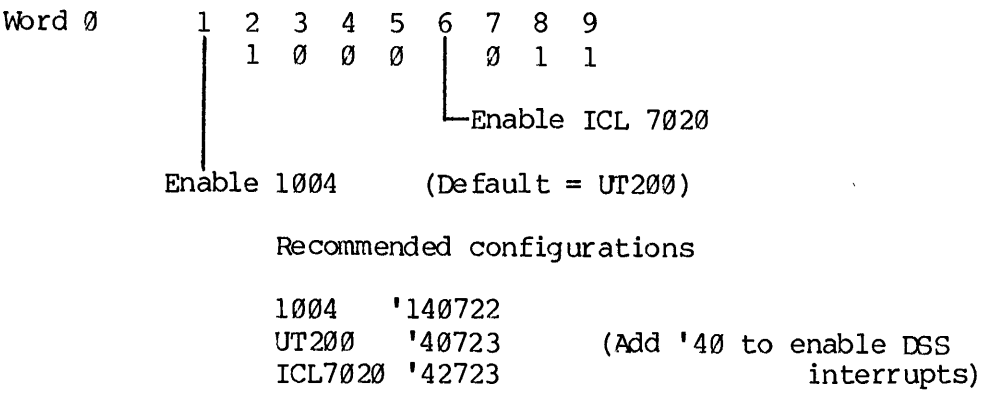


Table 20-7. Key = 3 Line Configuration Control Block (5651, bits 1-9).

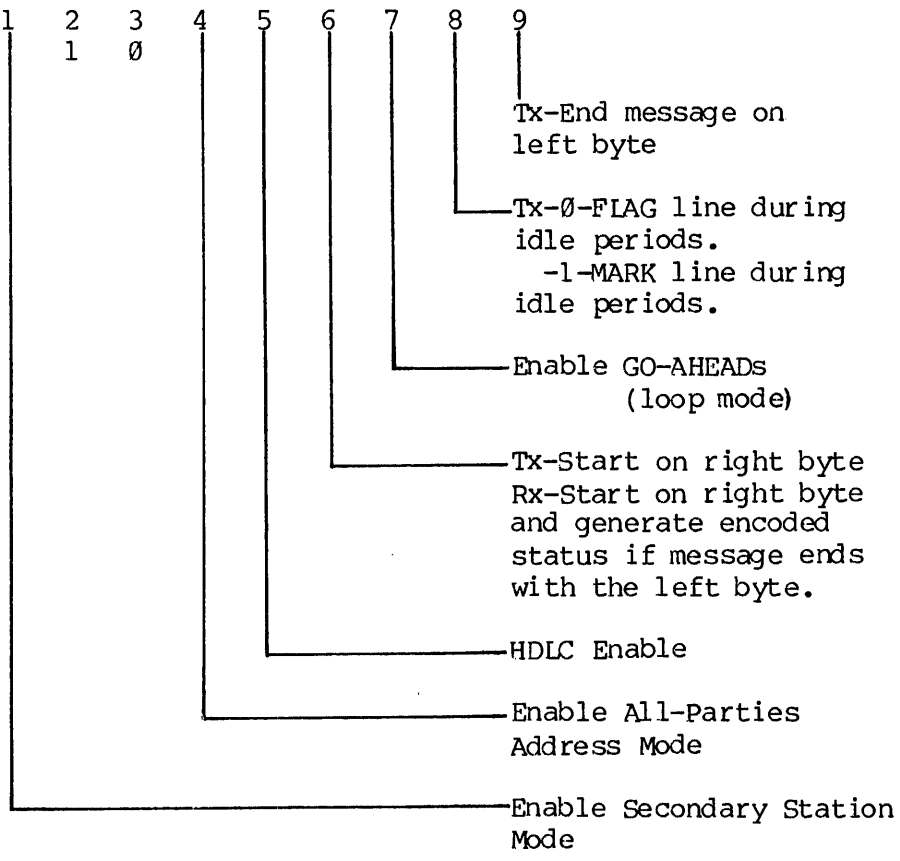
5651
ICL7020/UT200/1004

```
Word 0      1  2  3  4  5  6  7  8  9
            |  0  0  0  0  |  0  1  1
                           |
                     Enable ICL7020
            |
Enable 1004      (Default = UT200)
```

Recommended Configurations

UNIVAC	'100722
UT200	'723 (Add '40 to enable DSS interrupts)
ICL7020	'2723

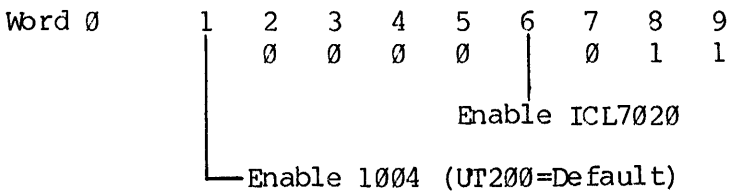
HDLC



Secondary Station Mode, HDLC mode, Loop mode, and
All Parties Address Mode are enabled on a line-pair basis only.

Table 20-8. Key = 3 Line Configuration Control Block (5652, bits 1-9).

5652
ICL7020/UT200/1004



Recommended Configurations

1004	'100722	
UT200	'723	(Add '40 to enable DSS interrupts)
ICL7020	'2723	

PACKET

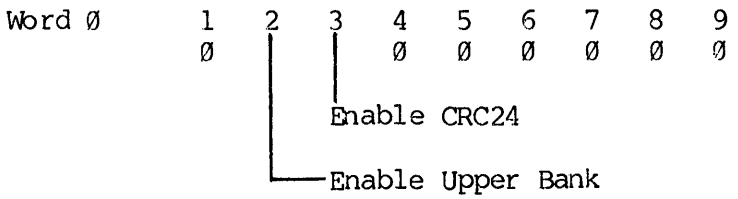
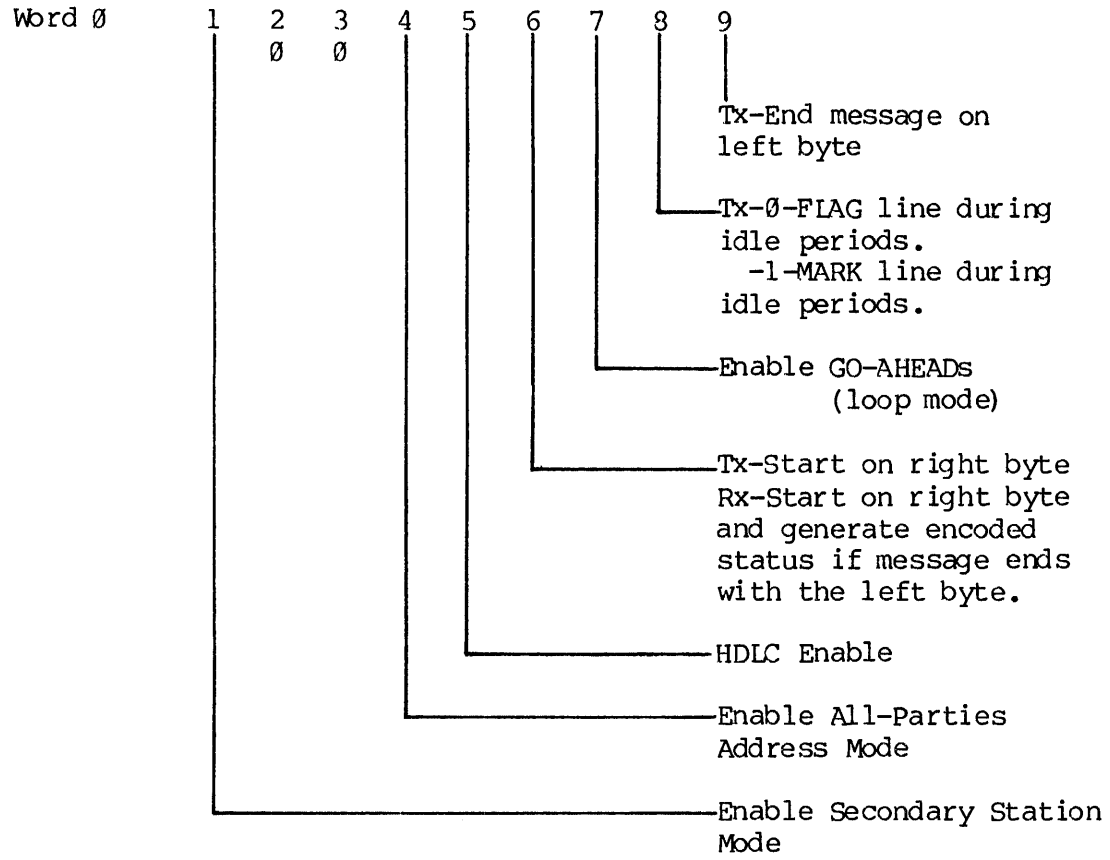


Table 20-9. Key = 3 Line Configuration Control Block (5653, bits 1-9).

5653
HDLC



Secondary Station Mode, HDLC mode, Loop mode, and
All Parties Address Mode are enabled on a line-pair basis only.

PACKET

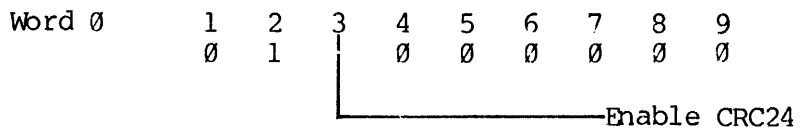
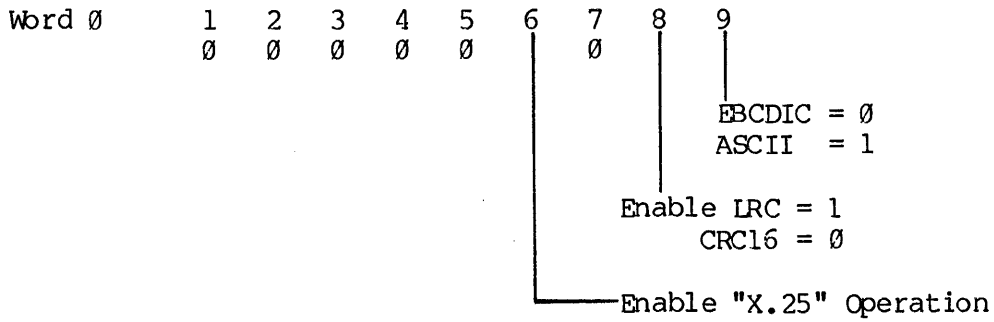


Table 20-10. Key = 3 Line Configuration Control Block (5654, bits 1-9)

5654
BISYNC



GRTS

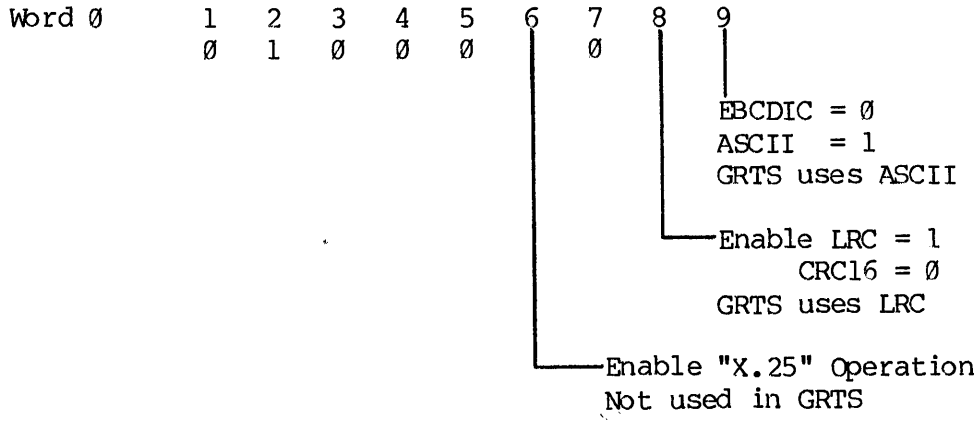


Table 20-11. Key = 3 Line Configuration Control Block (words 1-4).

Word 1	Word configuration - Transmitter Bit settings as for Word 0.
Word 2	Special character (OTA '00 : Function '10) Bits 7-8 : 00 Character 1 01 Character 2 10 Character 3 11 Character 4 Bits 9-16 : Character
Word 3	Special Character Bit settings as for Word 2
Word 4	Clock selection 0 Reset internal clock to default 9.6 Kbps 1 Switch internal clock to 62.5 Kbps

Table 20-12. Key=4 Data Set Control Bits (OTA '00:Function '00)

Bit 13	:	Not used
Bit 14	:	Speed Select
Bit 15	:	Request to send (RTS)
Bit 16	:	Data Terminal Ready (DTR)

Table 20-13. Key=5 Receive/Transmit Enable (OTA '00:Function '15)

Word 0	Bit 11	:	Select internal as receive clock
	Bit 12	:	Select internal as transmit clock
	Bit 13-14	:	00 Normal (transmit out, receive in) 01 Loop full duplex (transmit out, receive in) 10 Echo full duplex (receive in, transmit out) 11 Loop half duplex (pair combinations must be: 1-2, 2-1, 3-4, 4-3)
	Bit 15	:	1=Enable/0=Disable transmitter
	Bit 16	:	1=Enable/0=Disable receiver
Word 1	Bit 16	:	1=Enable transmitter 0=Enable receiver

Note

Transmitter and Receiver must be enabled/disabled separately.

ASYNCHRONOUS CONTROLLERS

The following describes the raw data mover for assigned AMLC lines. Refer to the System Administrator's Guide for the AMLC command, and how to assign AMLC lines.

▶ T$AMLC

CALL T$AMLC (line, LOC(buffer), ccnt, key, status)

line	Octal line number.
LOC(buffer)	Address of user's buffer.
ccnt	Character count (max is 80).
key	1 Input ccnt characters. 2 Input ccnt characters or until new line character, whichever occurs first. Return actual number of characters read in status(1). 3 Output ccnt characters. 4 Return number of characters in input buffer in status(1) and state of carrier in status(2). status(2) = 0 if carrier, not 0 if no carrier. 5 Return zero in status(1) if not room in output buffer for ccnt characters or one if there is room. State of carrier is also returned in status(2).
status	Two word status vector, described in key.

SECTION 21

REAL-TIME AND SYNCHRONIZATION SUBROUTINES

REAL-TIME AND INTERUSER COMMUNICATION FACILITIES

PRIMOS supports user applications that have real-time requirements or that need to synchronize execution with other user programs. Part of this support is the ability to modify the priority and timeslice duration of any user via the CHAP command. Program support for real-time applications and interuser synchronization is in the form of a set of subroutines that provide access to the Prime 400's semaphore primitives (WAIT and NOTIFY) and to internal timing facilities.

USER SEMAPHORES AND TIMERS

Internal to PRIMOS is an array of 64 semaphores reserved for the use of user processes. In the subroutines described below, all reference to a user semaphore is by the index of the semaphore, an integer from 1 to 64. Other than ensuring a valid semaphore number, PRIMOS makes no stipulations for semaphore use such as which users can access which semaphores, etc. Allocation and cooperative use of the semaphores is strictly under user control.

Of the 64 user semaphores, up to 15 can be used at any time as timed semaphores, that is, semaphores that are periodically notified by the system clock process (see the SEM$TN routine). Again, allocation of timed semaphores is on a first-come first-served basis, and nothing is done to prevent incorrect use of a timed semaphore.

Unless a user has inhibited quits from the user terminal, the typing of CONTROL-P or BREAK while a user process is waiting on a user semaphore, causes that semaphore to be notified and the process to enter command mode. The START command then causes control to return to the point following the call that resulted in the wait.

▶ SEM$DR

SEM$DR sets the value of a semaphore to zero.

CALL SEM$DR(semnum,code)

semnum The number of the user semaphore (1-64) to be drained. (INTEGER)

code An integer variable to be set to the return code. (INTEGER) Its value may be:

Ø = Request accepted.
E$BPAR = Invalid semaphore number.

If the value of the semaphore is negative (outstanding notifies), it is set to zero. (Other access to the semaphore is inhibited between the testing and setting to zero.) If the value of the semaphore is positive, 'value' notifies are executed, thus releasing a number of processes equal to the original value of the semaphore. Note that other user processes may wait on the semaphore in the middle of the notify loop. In this case, if the processes just entering the wait list have priority equal to or less than those already waiting, they will be left on the wait list at the conclusion of the SEM$DR call. If the new processes have priority higher than those already waiting, they will be immediately notified and some of the original processes will be left on the wait list.

▶ SEM$NF

SEM$NF notifies a specified user semaphore.

CALL SEM$NF(semnum,code)

semnum The number of the user semaphore (1-64) to notify. (INTEGER)

code An error or status code returned. (INTEGER) Its value may be:

Ø = Request accepted
E$BPAR = Invalid semaphore number
E$SEMO = too many notifies.

If the semaphore number is valid, the current value of the semaphore is checked. If the value is less than -32767, the code E$SEMO is generated, meaning that 32767 notifies have been issued with no

corresponding waits. (This is to prevent the semaphore value from wrapping to a positive value.)

▶ SEM$TN

SEM$TN establishes a user accessible "clock".

CALL SEM$TN(semnum,interval-1,interval-2,code)

semnum	The number of the user semaphore (1-64) to be used as a timer. (INTEGER)
interval-1	A variable that contains an interval in milliseconds to be awaited until the first notify of the semaphore. (INTEGER*4)
interval-2	A variable that, if non-zero, specifies an interval in milliseconds between all subsequent notifies. (INTEGER*4)
code	An error or status code returned. (INTEGER) Its value may be: Ø = Request accepted E$BPAR = Invalid semaphore number E$NTIM = No available timers.

SEM$TN first checks the validity of semnum and ensures that there is an available timer (i.e., that there are not already 15 active timer semaphores). A control block is then constructed and brought to the attention of the internal PRIMOS clock process. The clock process subsequently decrements interval-1 (in the control block) every tenth of a second until it reaches zero (or becomes negative, if the interval is not a multiple of 1ØØ msecs.). The specified user semaphore is then notified. If interval-2 is not Ø, interval-1 is then replaced with interval-2 (in the control block). The semaphore will then be notified every interval-2 msecs, until the timer is deactivated by a call to SEM$TN with an interval-1 value of Ø (INTEGER*4). If interval-1 is Ø, a search for the specified timer is made, and E$NTIM is returned if the timer is not found. If the timer is found, interval-1 is set to zero. This effectively deactivates the timer and releases it for other SEM$TN calls.

SEM$TN does not suspend execution of the user process. Execution is suspended only when SEM$WT is called and the specified interval has not expired. While the timed semaphore is in operation, it can be specified in calls to SEMWT, SEMNF, SEM$TS, or SEM$DR.

If QUIT is typed while a timed semaphore is active, notifies will accumulate while in command mode.

▶ SEM$TS

SEM$TS obtains the current count of waits/notifies

integer=SEM$TS(semnum,code)

integer — An integer variable set to the current value of the semaphore. (INTEGER)

semnum — The number of the user semaphore (1-64) to be tested. (INTEGER)

code — An error or status code returned. (INTEGER) Its value may be:

Ø = Request accepted
E$BPAR = Invalid semaphore number.

The current value of the semaphore is returned. If positive, the value indicates the number of processes currently waiting on the semaphore. If negative, the value reflects the number of outstanding notifies. Thus, in all cases,

value = waits - notifies

Note

The test operation is not interlocked against other access to the semaphore. Other processes issuing WAITs or NOTIFYs may change the value of the semaphore at any time after SEM$TS has returned the value of the semaphore.

▶ SEM$WT

SEM$WT enters the waitlist of specified sempahore.

CALL SEM$WT(semnum,code)

semnum — The number of the user semaphore (1-64) on which to wait. (INTEGER)

code — An integer error or status code returned. (INTEGER) Its value may be:

Ø = Request accepted
E$BPAR = Invalid semaphore number.

If the semaphore number is valid, a WAIT instruction is executed to place the user process on the wait list of the specified semaphore. The user process re-enters the ready list when the semaphore is notified or when QUIT is typed at the user terminal. If the semaphore has already been notified when SEM$WT is called, the WAIT instruction is a no-op, and control returns immediately.

▶ SLEEP$

SLEEP$ suspends execution of a user process.

CALL SLEEP$(interval)

interval — A variable containing the interval in milliseconds for which execution is to be suspended. (INTEGER*4)

Execution of the user process is suspended for the specified interval. An interval <Ø will result in an effective no-op. A QUIT and START from the user terminal will cause immediate return from the SLEEP$ call.

Part VI
Library Management

SECTION 22

LIBRARY MANAGEMENT

This section describes the Binary Editor (EDB) and LIBEDB. EDB is used to create and modify libraries. LIBEDB is used once a library is created to decrease loading time. Both of these programs operate on object text blocks generated by Prime language translators (FTN, COBOL, PMA, RPG). These objects text blocks form the input to LOAD and SEG. The term 'loader' is used to identify both programs.

LIBEDB

This program is used for editing bypass information into library files. The loader uses the bypass information to skip an unnecessary routine efficiently instead of reading and discarding all the unwanted object text. Depending on the size and number of unnecessary routines in a library, the loader may process library files up to 50 percent faster if they have been processed by LIBEDB.

LIBEDB is maintained as the run file *LIBEDB in the UFD 'LIB'. It should be used on a library file after its creation and after each time that the library is edited with the Binary Editor. The loader is capable, however, of handling a library which is not, or is only partially, processed by LIBEDB.

Since it is expected that LIBEDB will be used fairly infrequently, the user/computer interaction is self-explanatory. LIBEDB asks for an input and output file name and for file type. In theory, a library with large routines will load faster if it is created as a DAM file. In practice, none of the regularly used libraries contain routines large enough to warrant creating the library as a DAM file instead of as a SAM file.

EDB

Start-up

EDB is started up by the following command:

```
EDB input-file [output-file]
```

Both the input and output file may be pathnames. The input file should be an existing library or the binary output of a Prime language translator. The output file is optional; if specified, a file of that name will be created if none exists. -ASR or -PTR instead of a file on the command line specifies a user terminal or paper tape reader/punch

respectively. If these are not included, a PRIMOS file is assumed.

EDB types ENTER and then waits for user commands.

Operation

EDB maintains a pointer to the input file. When EDB is initialized, or after a TOP or NEWINF command, the pointer is at the top of the input file. The pointer can be moved by the FIND command to the start of a module. A module is identified by its subprogram or entry point name. After a COPY command (which copies blocks from the input to output file), the pointer is positioned to the module following the module copied.

Command Summary

EDB responds to the following commands, listed in alphabetical order. Commands may be abbreviated to the underlined letters. Items enclosed in brackets are optional.

BRIEF

Inhibits printout of subroutine names and entry points as they are encountered in the input file by EDB. (See TERSE and VERIFY.)

COPY {name, <SFL>, or <RFL> / ALL}

Copies to the output file, all main programs and subroutines from the pointer to (but not including) the subroutine called name or containing name as an entry point. If name is not encountered or COPY ALL is specified, EDB copies to the end of the input file and types .BOTTOM. on the terminal. The pointer moves past the last copied item.

FIND {name, <SFL>, or <RFL> / ALL}

Moves the pointer to module of the input file containing a subroutine called name or containing name as an entry point. If name is not found, the pointer is moved to the end of the input file and .BOTTOM. is typed on the terminal. In the VERIFY mode, the FIND ALL command can be used to print all subroutines and entry names in the input file.

INSERT pathname

Copies all modules of pathname to the output file. The pointer to the original input file is unchanged.

NEWINF pathname

Closes the current input file and opens pathname as the new input file. The pointer is positioned to the beginning of pathname.

OPEN

Closes the current output file and opens pathname as the new output file.

QUIT

Closes all files and exits to PRIMOS.

REPLACE (name) (pathname)

Replaces the object module containing (name) as an entry point by all modules of pathname.

RFL

Writes a reset-force-load flag block to the output file. All libraries begin with an RFL. This block places a loader in library mode; only those modules that are referenced are loaded. RFL mode is in effect until the loader encounters an SFL block.

SFL

Writes a set-force-load flag block to the output file. This block places a loader in force-load mode; all subsequent modules are loaded, whether or not they are called. SFL mode is in effect until the loader encounters an RFL block. A library file should be terminated by an SFL block.

TERSE

Places the editor into TERSE mode. Only the first entry point name of each module encountered by EDB is printed on the terminal. (See BRIEF, VERIFY)

TOP

Moves the pointer to the top of the input file.

VERIFY

Places EDB into VERIFY mode. All subroutine names and entry points, as they are encountered by EDB, are printed on the terminal. EDB is initialized in the VERIFY mode. (See BRIEF and TERSE)

The following commands are outmoded but are included for the sake of completeness:

ET

Writes an end-of-tape mark on the output file ('223, '223 on paper tape; zero word on disk). Writing an ET to disk causes the loader to ignore the remainder of the file.

GENET [G]

Copies the subroutine to which the pointer is currently positioned and follows it with an end-of-tape mark. The pointer moves to the next subroutine. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied, each followed by an end-of-tape mark. When the bottom of the input file is encountered, .BOTTOM. is printed on the terminal.

OMITET [G]

Copies the subroutine to which the binary location pointer is currently positioned. The pointer moves to the next subroutine. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied. When the bottom of the input file is encountered, .BOTTOM. is printed on the terminal.

EDB Error Messages

EDB prints ENTER to show that it is ready to accept commands. Most errors in command string input cause EDB to print a question mark (?). Other messages include:

BAD OBJECT FILE
usually a source file

BAD PARAMETERS
fatal

ERROR WHILE WRITING
fatal

EXAMPLES

Creating a Library

The following example creates a library from the files B_FILE1, B_FILE2, B_FILE3, and B_FILE4. Each file contains a single module although B_FILE1 and B_FILE2 contain multiple entry points.

The following terminal output shows the EDB commands to list the entry points of each file, plus the commands necessary to combine them into a library file, LIBEXP. Example:

```
OK, EDB B_FILE1
GO
ENTER, F ALL
ENT1A ENT1B ENT1C
.BOTTOM.
ENTER, NEWINF B_FILE2
ENTER F ALL
ENT2D ENT2E
.BOTTOM.
ENTER, NEWINF B_FILE3
ENTER, F ALL
ENT3G
.BOTTOM.
ENTER, NEWINF B_FILE4
ENTER, F ALL
ENT4H
.BOTTOM.
ENTER, OPEN LIBEXP
ENTER, NEWINF B_FILE1
ENTER, RFL
ENTER, C ALL
ENT1A ENT1B ENT1C
.BOTTOM.
ENTER, I B_FILE2
ENTER, I B_FILE3
ENTER, I B_FILE4
ENTER, SFL
ENTER, QUIT
```

After a library is created, LIBEDB can be run on it to speed its loading time.

Listing Entry Points

Notice the difference between the terminal output in VERIFY and TERSE modes. ENT1A, ENT1B, and ENT1C are all entry points of the first module. In TERSE mode, only ENT1A is listed. Example:

```
OK, EDB LIBEXP
GO
ENTER, F ALL
ENT1A ENT1B ENT1C ENT2D ENT2E ENT3G ENT4H
.BOTTOM.
ENTER, TOP
ENTER, TERSE
ENTER, F ALL
ENT1A ENT2D ENT3G ENT4H
.BOTTOM.
ENTER, QUIT
```

Replacing an Object Module in the Library

The library file, LIBEXP, created above is edited to replace the module containing entry point ENT3G with the module in B_NFILE3 containing entry points ENT3F and ENT3G. The output file is LIBNEW.

```
OK, EDB B_NFILE3
GO
ENTER, F ALL
ENT3F ENT3G
.BOTTOM.
ENTER, Q

OK, EDB LIBEXP LIBNEW
GO
ENTER, R ENT3G B_NFILE3
ENT1A ENT1B ENT1C ENT2D ENT2E ENT3G
ENTER, C ALL
ENT4H
.BOTTOM.
ENTER, Q

OK, EDB LIBNEW
GO
ENTER, F ALL
ENT1A ENT1B ENT1C ENT2D ENT2E ENT3F ENT3G ENT4H
.BOTTOM.
ENTER, Q
```

Part VII
Condition Mechanism Subroutines

SECTION 23

CONDITION MECHANISM SUBROUTINES

INTRODUCTION

This section describes the subroutines used in the implementation of the condition mechanism. A condition is an unscheduled software procedure call (or block activation) resulting from an "unusual event". Such an "unusual event" might be a hardware-defined fault, an error situation which cannot be adequately defined to the subroutine, or an external event such as a QUIT from the user's terminal. The condition mechanism has been created to:

- Provide a consistent and useful means for system software to handle error conditions
- Provide the capability to handle error conditions without forcing a return to command level
- Provide support for the condition mechanism of ANSI PL/I

CREATING AND USING ON-UNITS

Condition handlers are called "on-units" which may be procedures or PL/I begin blocks. A begin block results from a PL/I <on statement> while a procedure results from the use of the subroutines:

MKONU$
MKON$F

The use of these subroutines is the only way to create an on-unit in a non-PL/I environment. All users are automatically protected by PRIMOS system on-units. When a condition is raised, the condition mechanism searches within the existing procedure for on-units for the specific condition. If none is found, but if an on-unit for the special condition ANY$ does exist, the ANY$ on-unit is selected as the default on-unit.

An on-unit may be invalidated by the PL/I <revert statement> or by using the subroutines:

RVONU$
RVON$F

The condition mechanism is activated whenever a condition is raised. A condition is raised implicitly by some exception being detected during regular program execution. A condition may be raised explicitly by the PL/I <signal statement> or by a call to the subroutines:

SIGNL$
SGNL$F

Every on-unit has the name of the condition it is handling. A condition name is a character string (up to 32 characters) and may represent a system defined condition if the name is one reserved for system use, or it may be a user-defined condition. The system-defined conditions are described later in this section.

On-Unit Actions

An on-unit has several options on action it may take. An on-unit may:

1. Perform application specific tasks (e.g., closing files, updating files).
2. Repair cause of condition and resume execution.
3. Decide that normal flow can be interrupted and program re-entered at "known point" by performing a nonlocal goto to some previously defined label.
4. Signal another condition.
5. Transfer process to command level.
6. Continue search for more on-units.
7. Run diagnostic routines.

FORTRAN Considerations

The use of on-units and of nonlocal gotos from FORTRAN is somewhat restricted, since there are no internal procedures or blocks. Therefore:

- FORTRAN on-units must be subroutines which, by definition, are not internal to the subroutine or main program creating the on-unit.
- Nonlocal gotos will work only to the previous stack level since the target statement label belongs to the caller of the subroutine performing the nonlocal goto.

A full function nonlocal goto requires that the target label identify both a statement and a stack frame of the program that contains the

statement. The subroutine MKLB$F will create a PL/I compatible label and the subroutine PL1$NL will perform a nonlocal goto to a specified target label. Labels produced by MKLB$F are acceptable to PL1$NL.

The PL/I interfaces utilize the PL/I datatype "character(*) varying". This datatype is not available in FORTRAN, but 1977 ANSI FORTRAN includes a datatype which is the equivalent of PL/I "character(*) nonvarying". Interfaces are provided which utilize the nonvarying character strings. These will not be as efficient as those using varying character strings. It is possible to simulate varying character strings in FORTRAN with an INTEGER*2 array in which the first element contains the character count, and the remaining elements contain the characters in packed format. For example:

```
PL/I
   dcl name char(5) varying static initial ('QUIT$');

FORTRAN
   INTEGER*2 NAME(4)
   DATA NAME/5, 'QUIT$'/
```

The subroutines are documented in PL/I, and therefore FORTRAN users must make a conversion between PL/I datatypes and FORTRAN datatypes. The following is a table for such a conversion:

PL/I	FORTRAN
char(n) var	INTEGER(((n+1)/2)+1)
char(n)	INTEGER((n+1)/2)
fixed bin(15)	INTEGER
fixed bin(31)	INTEGER*4
label	REAL*8
entry variable	REAL*8
ptr options (short)	INTEGER*4
bit(n)	INTEGER (1<=n<=16)

Default On-Unit

The default on-unit, ANY$, may be created to intercept any condition that might be activated during a procedure. (The ANY$ on-unit is created by a PL/I <on-statement> or a call to MKONU$ or MKON$F.)

When a condition is raised, the condition mechanism first searches for an on-unit for the specific condition. If a specific on-unit exists, it is selected, but if none exists and an ANY$ on-unit exists, it is selected.

User programs should avoid the use of the ANY$ on-unit. If used, a user ANY$ on-unit should not attempt to handle most system-defined conditions, and should pass them on by simply returning. Whenever an ANY$ on-unit is invoked, the continue switch is set and the user ANY$ on-unit must return with the continue switch still set. Failure to do so can cause problems with PRIMOS.

The continue switch (cflags.continue_sw) is used to indicate to the condition mechanism whether the on-unit that was just invoked (or any of its dynamic descendants) wishes the backward scan of the stack for on-units for this condition to continue upon the on-units return. The subroutine CNSIG$ is used to request that the switch be turned on. This switch is cleared before each on-unit (except ANY$) is invoked.

CONDITION MECHANISM SUBROUTINES

▶ SIGNL$

SIGNL$ is called to signal a specific condition. The stack is scanned backwards to find an on-unit for this condition or a default (ANY$) on-unit.

```
dcl signl$ entry (char(*) var, ptr, fixed bin, ptr, fixed bin,
                  bit(16) aligned);

CALL SIGNL$ (condition_name, ms_ptr, ms_len, info_ptr,
             info_len, action);
```

condition_name	Name of condition to be signalled.
ms_ptr	Pointer to stack frame header structure defining the machine state at the time the specific condition was detected. If ms_ptr is null, a pointer to the condition frame header, produced by this call to SIGNL$ will be used.
ms_len	Length (in words) of the structure named in ms_ptr. Not examined if ms_ptr is null.
info_ptr	Pointer to structure containing auxiliary information about the condition. If no auxiliary info is available, info_ptr should be null.
info_len	Length (in words) of structure in info_ptr. Not examined if info_ptr is null.

action — Defines action to be taken.

```
dcl 1 action,
      2 return_ok bit(1),
      2 inaction_ok bit(1),
      2 crawlout bit(1),
      2 mbz bit(12);
```

return_ok = '1'b if on-unit is to be allowed to return.

inaction_ok = '1'b if on-unit may return without taking corrective action and still expect "defined" results. (return_ok must also be '1'b.)

crawlout = '1'b if call to SIGNL$ is result of crawlout. Should never be set by user.

specifier = '1'b to signal PL/I I/O(PLIO) condition. User program should not use.

mbz = Must be zero.

▶ SGNL$F

SGNL$F signals a specific condition and supplies optional auxiliary information. SGNL$F is the FORTRAN equivalent of SIGNL$ but is less efficient in use of time and space.

```
CALL SGNL$F (cname, cnamel, msptr, mslen, infopt, infoln, flags)
INTEGER*2 cname(--), cnamel, mslen, infoln, flags
INTEGER*4 msptr, infopt
```

cname — Name of condition to be signalled.

cnamel — Length of cname in characters.

msptr — Pointer to location of stack frame header describing machine state at time the specific condition was detected. User does not usually know this information and should pass the null pointer value of :1777600000.

mslen — Length (in words) of stack frame header.

infopt — Pointer to location of user-supplied auxiliary information array. If no information supplied user should pass null pointer. (:1777600000).

infoln Length, in words, of array pointed to by infopt.

flags Action array specifying control action

BIT

1=1 On-unit may return

2=1 On-unit may return without taking action

3=1 Call is result of crawlout - should never be set by user

4=1 Signal PLIO condition - User should not set

5-16 Must be zero

▶ CNSIG$

CNSIG$ instructs the condition mechanism to continue scanning for more on-units for the specific condition that was raised after the calling on-unit returns. CNSIG$ is called when an on-unit has been unable to completely handle the condition. The continue switch is set in the most recent condition frame.

```
dcl  cnsig$   entry (fixed bin);

CALL cnsig$   (status);
```

status Standard system error code. Will be non-zero only if there was no condition frame found in the stack.

Note

The continue switch is automatically set whenever an ANY$ on-unit is invoked. Therefore, an ANY$ on-unit need not issue a call to CNSIG$ to continue to signal.

▶ MKLB$F

MKLB$F converts a FORTRAN statement label or an integer variable with a statement label value, into a PL/I compatible label value. This label value can then be used with a call to the subroutine PL1$NL, to perform a full function nonlocal goto in a FORTRAN program.

```
CALL MKLB$F (stmt, label)
INTEGER*2 stmt
REAL*8 label
```

stmt — Variable to which a FORTRAN statement number has been assigned by an ASSIGN statement, or is a statement number constant in the format $XXXXX.

label — Contains PL/I compatible label value for stmt after call to MKLB$F.

▶ PL1$NL

PL1$NL performs a full function nonlocal goto to the statement identified in the call. Label values created by MKLB$F are suitable arguments for PL1$NL.

```
CALL PL1$NL (label)
REAL*8 label
```

label — PL/I compatible label value.

▶ MKONU$

MKONU$ creates an on-unit for a specific condition or creates a default on-unit for the ANY$ condition. MKONU$ cannot be called from the PL/I-G version of PL/I. Instructions for PL/I-G users are in description of MKON$F.

```
dcl    mkonu$  entry (char(*) var, entry);

CALL mkonu$ (condition_name, handler);
```

condition_name — Name (no trailing blanks) of condition for which on-unit will be created. Any previous on-unit for this condition within the activation, will be over-written.

handler — Entry value representing on-unit procedure to be invoked when condition name is raised and this activation is reached in the stack scan. Since MKONU$ does not save the display pointer associated with on-unit entry, the entry value must be external or declared in the block calling MKONU$. (An entry constant declared in the block containing the call to MKONU$ will satisfy these restrictions).

Note

The stack frame of the caller is grown, if necessary, to add the descriptor block for the new on-unit.

The caller must guarantee that the storage occupied by condition name will not be freed until the caller returns, or the activation is aborted by a nonlocal goto. For PL/I callers, this restriction means condition name may not be a constant.

▶ MKON$F

MKON$F creates an on-unit for a specific condition and is intended for the FORTRAN or PL/I-G user. The function is the same as MKONU$ but is substantially less efficient in terms of stack space and execution time. The FORTRAN usage is:

```
CALL MKON$F (cname, cnamel, unit)
EXTERNAL UNIT
INTEGER*2 CNAME(--), CNAMEL
```

cname — Array containing name of condition for which on-unit is to be created.

cnamel — Length (in characters) of cname

unit — External subroutine which will be the on-unit handler. This subroutine is called with

```
CALL UNIT (CP)
INTEGER*4 CP
```

CP is a pointer to the condition frame header (cfh) that describes the condition.

Note

FORTRAN programs using MKON$F must include the specification statement "STACK HEADER 34" and must be compiled with the SPO option. This will reserve the stack space for on-unit information. If MKONU$ is used, its SHORTCALL specification will reserve the space.

Cname and cnamel may be over-written by the caller once MKON$F has returned, since they are copied into a stack frame extension.

The PL/I-G usage is:

```
dcl  MKON$F entry (char(*), fixed bin, ptr);
dcl  ev entry variable;
dcl  p ptr based;

ev = handler
CALL MKON$F (condition_name,cond_name_len,addr(ev)->p->p);
```

condition_name — Name of condition for which on-unit is to be created.

cond_name_len — Length of condition name in characters.

handler — Entry value representing the routine to be invoked as the on-unit. Restrictions described under MKONU$ apply here as well.

Note

This different calling sequence is required because FORTRAN and PL/I-G differ in the way they represent entry (subroutine) values.

▶ RVONU$

RVONU$ disables (reverts) an on-unit for a specific condition. Once disabled, the on-unit will be ignored during stack-frame scanning. The on-unit may be re-instated only by another call to MKONU$ or MKON$F. A call to RVONU$ affects only on-units within its own activation.

```
dcl    rvonu$ entry (char(*) var);
CALL   rvonu$ (condition_name);
```

condition_name — Name of condition for which the on-unit is to be disabled.

Note

There is no effect if an on-unit does not exist for the named condition, or if the on-unit has already been disabled. A call to RVONU$ will not affect on-units in any other activation.

▶ RVON$F

RVON$F disables an on-unit for a specific condition. Its effect is identical to RVONU$ but is designed for the FORTRAN user, and is less efficient in terms of space and execution time.

```
CALL RVON$F (cname, cnamel)
INTEGER*2 CNAME(--), CNAMEL
```

cname Name of condition for which the on-unit is to be disabled.

cnamel Length (in characters) of cname

Note

There is no effect if an on-unit does not exist for the named condition, or if the on-unit has already been disabled.

SYSTEM-DEFINED CONDITIONS

The following are the standard system-defined condition names, with the meaning of each condition and, where available, the information structure produced by each condition. The standard PL/I information structure is:

```
dcl 1  info based,
       2 file_ptr, ptr options (short), /*PL/I file control block*/
       2 info_struct_len fixed bin,     /*Length in words of*/
                                        /*structure*/
       2 oncode_value fixed bin,        /*unique error code */
       2 ret_addr ptr options (short),  /*Points to statement causing*/
                                        /*error.*/
```

The condition frame header (cfh), stack frame header (sfh), fault frame header (ffh) and on-unit descriptor block formats are included at the end of this section.

If not stated otherwise, the system default on-unit for each condition prints an appropriate diagnostic message on the user's terminal, and calls a new command level.

▶ ACCESS_VIOLATION$

A CPU instruction which violated the access rules of the processor, has been attempted.

ffh.fault_type	'44'B3
ffh.fault_addr	Improperly accessed virtual address
ffh.ret_pb	Points to instruction causing violation

▶ ANY$

ANY$ is a special condition name used for default on-units. The condition frame header will describe the actual specific condition. There is no separate condition frame header for the condition ANY$ unless ANY$ has been explicitly raised by a call to SIGNL$ (not recommended).

▶ ARITH$

An arithmetic exception has occurred.

ffh.fault_type	'5Ø'B3
ffh.fault_code	Hardware-defined Exception Code which partially identifies cause of fault.
ffh.ret_pb	Points to next instruction to be executed upon return. There is no way to obtain a pointer to the faulting instruction.

The static mode (see discussion of static mode under Recursive Mode Software later in this section) default on-unit for this condition will simulate PRIME 3ØØ fault handling for Arithmetic Exception if the appropriate word of segment '4ØØØ (see the System Architecture Guide for the exact location of the word), is non-zero. If not in static mode, or if PRIME 3ØØ vector word is zero, the standard handler for this condition will resignal the ERROR condition (described below).

▶ BAD_NONLOCAL_GOTO$

The nonlocal goto processor has been given an invalid display pointer. The display (stack) pointer may actually be invalid or the target activation may previously have been cleaned-up, or the user's stack may have been overwritten. Information structure:

```
dcl  1 info based,
       2 target_label,                          /*Target of nonlocal*/
                                                /*goto*/
       2 ptr_to_nlg_call ptr options (short),   /*Pointer to nonlocal*/
                                                /*goto call*/
       2 caller_sb ptr;                         /*Pointer to calling*/
                                                /*stack frame*/
```

▶ BAD_PASSWORD$

This condition is raised by the ATCH$$ subroutine (Section 4) when an attempt is made to attach to a directory with an incorrect password.

▶ CLEANUP$

The nonlocal goto processor raises this condition prior to actually unwinding the stack. The on-unit for this condition should return unless it encounters a fatal error. Calls to CNSIG$ from a CLEANUP$ on-unit have no effect; the search for on-units continues until the target activation is reached.

▶ ENDFILE (file) PL/I

This condition is raised when end-of-file is encountered while reading a PL/I file with PL/I input/output statements. The value of the onfile() builtin function identifies the file involved.

info.oncode_value	Undefined
info.file_ptr	Identifies file

The default on-unit for this condition prints a diagnostic and resignals the ERROR condition with an info.oncode_value of 1044.

▶ ENDPAGE (file) PL/I

End-of-page has been encountered while writing a PL/I file with PL/I input/output statements. The value of the onfile() builtin function identifies the file for which end-of-page was encountered. The default on-unit for this condition performs a "put skip" on the file and returns.

info.oncode_value	Undefined
info.file_ptr	Identifies file

▶ ERROR PL/I

This is the default on-unit for most PL/I-defined conditions. Many I/O and conversion operations result in the raising of the ERROR condition. Each distinct error has been assigned a unique info.oncode_value. The standard PL/I info structure is described above.

The default on-unit prints a diagnostic using the value of info.oncode_value, and calls a new command level unless the error is one of the arithmetic errors that is handled "without comment". With those errors, the appropriate action is taken and the on-unit returns to the point of interruption.

▶ ERRRTN$

A non-ring-Ø call to ERRRTN has been made with an ERRRTN SVC or a call to ERRPR$. The default on-unit for this condition simulates a call to EXIT. This condition is used by PRIMOS to ensure the correct operation of user programs and therefore, should not be handled by user programs.

▶ EXIT$

The EXIT subroutine (Section 5) has been called directly or via an EXIT SVC. This condition should not be handled by user programs since it is used by the Source Level Debugger (DBG) to monitor the execution of STATIC MODE programs.

▶ ILLEGAL_INST$

An attempt has been made to execute an illegal instruction.

ffh.fault_type	'4Ø'B3
ffh.ret_pb	Points to illegal instruction

▶ ILLEGAL_ONUNIT_RETURN$

An on-unit has attempted to return but return was disallowed by the procedure that raised the condition. The information structure is a standard-format condition frame header that describes the condition whose on-unit has illegally attempted to return.

▶ ILLEGAL_SEGNO$

Reference has been made to a virtual address with an out-of-bounds segment number.

```
ffh.fault_type    '60'B3
ffh.ret_pb        Points to instruction causing error
ffh.fault_addr    Virtual address in error
```

▶ KEY (file) PL/I

This condition is raised when reading a PL/I file with a non-existent key or writing with a key that already exists. The onfile() builtin function identifies the file. The onkey() builtin function contains the key in error. The default on-unit prints a diagnostic and resignals the error condition with info.oncode_value of 1045.

```
info.oncode_value Undefined
info.file_ptr     Identifies file
```

▶ LINKAGE_FAULT$

An indirect pointer (IP) with a valid unsnapped dynamic link has been referenced but the desired entry point was not found in the dynamic link tables.

```
ffh.fault_type    '64'B3
ffh.fault_addr    Points to IP
ffh.ret_pb        Points to instruction causing error.
```

Info structure:

```
dcl 1 info based,
      2 entry_name char(32)var;   /*Name of entry point not found*/
```

▶ LISTENER_ORDER$

This condition is used internally by the command loop to manage its recursion. Users should never create on-units for this condition and user default on-units (ANY$) should always pass this condition on by returning.

▶ NO_AVAIL_SEGS$

Reference has been made to a virtual address that refers to a segment not yet created. The system has no free page tables to assign to the segment. If the on-unit for this condition returns, the reference will be retried and will succeed if a segment has become available.

ffh.fault_type	'60'B3
ffh.ret_pb	Points to instruction causing error
ffh.fault_addr	Virtual address referencing segment

▶ NONLOCAL_GOTO$

A nonlocal goto is about to occur. This condition is signalled by PL1$NL immediately before setting up the stack unwind and therefore prior to any call of CLEANUP$ on-units. The default on-unit for this condition simply returns. Any user-written procedure should return (without setting the continue-to-signal) in order to allow the goto to proceed. Information structure:

```
dcl  1 info based,
       2 target_label,                          /*Target of nonlocal*/
                                                /*goto*/
       2 ptr_to_nlg_call ptr options (short),/*Pointer to PL1$NL*/
                                                /*call*/
       2 info.caller_sb;                        /*Pointer to stack frame*/
                                                /*requesting nonlocal*/
                                                /*goto.*/
```

▶ NULL_POINTER$

A reference has been made through an indirect pointer or base register whose segment number is '7777'B3. This is considered to be a reference through a null pointer, although user software should always use the single value 7777/0 for the null pointer. The default on-unit resignals the ERROR condition.

ffh.fault_type	'60'B3
ffh.ret_pb	Points to instruction making reference
ffh.fault_addr	Null pointer through which reference was made.

▶ OUT_OF_BOUNDS$

Reference has been made to a page of some segment for which no main memory or backing storage has been allocated, and allocation is not permitted.

ffh.fault_type	'1Ø'B3
ffh.ret_pb	Points to instruction making illegal reference
ffh.fault_addr	Illegal virtual address

▶ PAGE_FAULT_ERR$

A valid virtual address has been referenced but because of a disk error, the page control mechanism has been unable to load the page into main memory. If the on-unit for this condition returns, the reference will be retried. If the disk read succeeds, the reference will be completed.

ffh.fault_type	'1Ø'B3
ffh.ret_pb	Instruction with illegal reference
ffh.fault_addr	Virtual address causing problems

▶ PAUSE$

A PAUSE statement has been executed in a FORTRAN program. This condition is used by PRIMOS to ensure the proper operation of the PAUSE statement, and should not be handled by user programs. The default on-unit prints no diagnostic, but calls a new command level.

▶ POINTER_FAULT$

Reference has been made through an indirect pointer (IP). The fault bit of the pointer is on, but the pointer did not appear to be a valid unsnapped dynamic link.

ffh.fault_type	'64'B3
ffh.fault_addr	Points to invalid IP
ffh.ret_pb	Points to instruction causing error

▶ QUIT$

The user has activated the quit button (Break key or Control-P) on the terminal. The default on-unit flushes the input and output buffers of the user's terminal, prints "QUIT" on the terminal and calls a new command level.

ffh.fault_type	'04'B3
ffh.ret_pb	Points to next instruction to be executed

▶ REENTER$

This condition is raised by the PRIMOS REENTER (REN) command and reenters a subsystem that has been temporarily suspended due to another condition (such as a QUIT signal).

▶ RESTRICTED_INST$

Attempt has been made to execute an instruction restricted to ring 0 procedures. Although some of these instructions in the I/O class can be simulated by ring-0, an instruction causing this condition to be raised, cannot be simulated.

ffh.fault_type	'00'B3
ffh.ret_pb	Points to instruction in error

▶ R0_ERR$

A ring 0 call to ERRPR$ or ERRRTN has been made because of the detection of a fatal error. The default on-unit prints no diagnostic but calls a new command level.

▶ STACK_OVF$

The process has overflowed one of its stack segments, but the condition mechanism was able to locate a stack on which to raise this condition. The static mode default on-unit will attempt to simulate the PRIMOS stack overflow fault, if the appropriate word of segment '4000 is non-zero (see the System Architecture Guide for more information). If this word is zero, or if no static mode program is being executed, standard default handling occurs.

ffh.fault_type	'54'B3
ffh.fault_addr	Last stack segment in stack that overflowed
ffh.ret_pb	Points to instruction causing error

▶ STOP$

A STOP statement has been executed in a higher-level-language program. This condition is used by PRIMOS to ensure the proper operation of the STOP statement in the various languages. This condition should not be handled by user programs. The default on-unit performs a nonlocal goto back to the command processor invoking the procedure which executed the STOP statement.

▶ SVC_INST$

An SVC instruction has been executed, but the system is unable to perform the operation. If the user is in "SVC virtual" mode, all SVC instructions raise this condition. For virtual SVC's, the static mode default on-unit will simulate PRIMOS III fault handling for the SVC fault, if the appropriate word of segment '4000 is non-zero (See System Architecture Guide for word location). If this word is zero, or there is no static mode program in execution, the standard defualt handler prints a diagnostic and calls a command level.

```
ffh.fault_type      '14'B3
ffh.ret_pb          Points to location following SVC.
```

Information structure:

```
dcl 1 info based,
      2 reason fixed bin;
```

Values of info.reason are:

1 Bad SVC operation code or bad argument
2 Alternate return needed but was zero
3 Virtual SVC handling in effect

▶ UNDEFINED_GATE$

The process has called an inner ring gate segment at an address within the initialized portion of the gate segment, but no legal gate is found at that address. This results from gate segments being padded to the next page boundary with "illegal" gate entries.

▶ UII$

The process has executed an unrecognized instruction that caused an Unimplemented Instructiion (UII) Fault, or the system UII handler detected an error in processing the valid UII.

ffh.ret_pb	points to next instruction
ffh.regs	is not valid

CRAWLOUT MECHANISM

An event known as a crawlout occurs whenever the Condition Mechanism reaches the end of an inner ring stack (a ring other than 3) without finding a selectable on-unit for the condition that has been raised. Note that a crawlout can occur even when the inner ring has an on-unit for the condition, if that on-unit signals another condition, or if the on-unit calls CNSIG$ and returns, causing a resumption of the stack scan. The scan for on-units resumes on the stack of the ring which invoked the inner ring. The outer ring receives a copy of the machine state at the time the condition was raised.

RECURSIVE MODE SOFTWARE

The Recursive Command Environment provides a fully recursive command processing loop that is also highly modular. The implementation of the new environment partitions system and user software into two categories - recursive mode and static mode.

Static mode software:

- allocates its own segments.
- manages its own stack.
- manages its own shared libraries' initialization.
- uses a "memory image" approach in which the program is reloaded each time it is called and therefore programs may not be recursively invoked from command level.

User on-units, any procedures they call, and all internal commands are recursive mode software and therefore have the following properties:

- use the system stack.
- terminate by returning level.
- do not attempt to initialize shared libraries.
- are not reloaded as memory image each time called.

A recursive mode procedure must terminate by returning, not by calling EXIT. Arguments for recursive mode commands are passed as parameters and are not obtained from some static buffer. Error information is passed by setting a return parameter (error code), printing an error message and returning, or by signalling a condition. The ERRRTN call must not be used and ERRPR$ may be used only with the immediate-return key.

The subroutines COMLV$ and CMLV$E are available to the user in the recursive environment.

▶ COMLV$

COMLV$ invokes a new listener level of the PRIMOS Command Loop. When the command loop returns, COMLV$ will return to its caller.

```
dcl comlv$ entry ();
CALL comlv$;
```

COMLV$ is used when there is no "command error" to report to the user.

▶ CMLV$E

CMLV$E invokes a new listener level of the PRIMOS Command Loop and is used when "command error" processing at the new command level is desired. The command input file (if any) is paused, command output to the terminal is forced on, QUITS are enabled and "ER" is used for the prompt message.

```
dcl cmlv$e entry ();
CALL cmlv$e;
```

Note

COMLV$ and CMLV$E switch stacks to the command processor stack, if the process was not already executing on that stack.

DATA STRUCTURE FORMATS

The data structures associated with the condition mechanism are described below. Any user program that uses these structures should examine the version number in the structure (if one is provided); if the format of a structure changes, the version number will be incremented. The user program can then take appropriate action if it is presented with structures of different formats.

The Condition Frame Header (CFH)

The following declaration shows the format of the Standard Condition Frame Header:

```
dcl    1 cfh based,  /* standard condition frame header */
           2 flags,
             3 backup_inh bit(1),
             3 cond_fr bit(1),
             3 cleanup_done bit(1),
             3 efh_present bit(1),
             3 user_proc bit(1),
             3 mbz bit(9),
             3 fault_fr bit(2),
           2 root,
             3 mbz bit(4),
             3 seg_no bit(12),
           2 ret_pb ptr,
           2 ret_sb ptr,
           2 ret_lb ptr options (short),
           2 ret_keys bit(16) aligned,
           2 after_pcl fixed bin,
           2 hdr_reserved(8) fixed bin,
           2 owner_ptr ptr options (short),
           2 cflags,
             3 crawlout bit(1),
             3 continue_sw bit(1),
             3 return_ok bit(1),
             3 inaction_ok bit(1),
             3 specifier bit(1),
             3 mbz bit(11),
           2 version fixed bin,
           2 cond_name_ptr ptr options (short),
           2 ms_ptr ptr options (short),
           2 info_ptr ptr options (short),
           2 ms_len fixed bin,
           2 info_len fixed bin,
           2 saved_cleanup_pb ptr options (short);
```

flags.backup_inh
will always be '0'b in a condition frame. It is used in regular call frames to control program counter backup on crawlout from an inner ring.

flags.cond_fr
identifies this frame as a condition frame, and will thus be '1'b.

flags.cleanup_done
is '1'b when this activation has been "cleaned up" by the procedure unwind_, which helps to effect nonlocal goto's. When this flag is set, the value of cfh.ret_pb no longer describes the return point of the activation; that information is available in cfh.saved_cleanup_pb.

flags.efh_present
will always be '0'b in a condition frame. It is used in a regular call frame to indicate that an extended stack frame header containing on-unit data is present.

flags.user_proc
identifies stack frames belonging to "non-support" procedures, and hence will be '0'b in a condition frame.

flags.mbz
is reserved and will be '0'b.

flags.fault_fr
will always be '00'b in a condition frame.

root.mbz
is reserved and must be '0'b.

root.seg_no
is the hardware-defined stack root segment number, and indicates which segment contains the stack root for the stack containing this fault frame.

ret_pb
points to the next instruction to be executed following the call to SIGNL$ that caused this condition to be raised, unless flags.cleanup_done is '1'b, in which case cfh.ret_pb will point to a special code sequence used during stack unwinds, and cfh.saved_cleanup_pb will contain the former value of cfh.ret_pb.

ret_sb
is the hardware-defined stack base of the caller of SIGNL$. Thus, this value also points to the previous stack frame on the stack.

ret_lb
is the hardware-defined linkage base of the caller of SIGNL$.

ret_keys
is the hardware-defined keys register of the caller of SIGNL$.

after_pcl
is the hardware-defined offset of the first argument pointer following the call to SIGNL$ that raised this condition.

hdr_reserved
is reserved for future expansion of the hardware-defined PCL/CALF stack frame header, of which the totality of cfh is a further extension.

owner_ptr
is reserved to point to the ECB of the procedure that owns this stack frame (usually SIGNL$).

cflags.crawlout
is '1'b if this condition occurred in an inner ring (a ring number lower than the ring in which the on-unit is executing), but could not be adequately handled there; otherwise it is '0'b.

cflags.continue_sw
is used to indicate to the condition mechanism whether the on-unit that was just invoked (or any of its dynamic descendants) wishes the backward scan of the stack for on-units for this condition to continue upon the on-unit's return. The subroutine CNSIG$ is used to request that cflags.continue_sw be turned on; user programs should NOT attempt to set it directly. This switch is cleared before each on-unit is invoked (except ANY$ on-units).

cflags.return_ok
is '1'b if the procedure that raised the condition is willing for control to be returned to it by means of the on-unit simply returning. If '0'b, an attempt by an on-unit for this condition to return will cause the special condition ILLEGAL_ONUNIT_RETURN$ to be signalled. Note, however, that the on-unit may return regardless of the state of cfh.cflags.return_ok if cfh.cflags.continue_sw has previously been set by a call to CNSIG$. This is because, in this case, the on-unit return does not cause a return to the procedure that raised the condition, but instead causes a resumption of the stack scan.

cflags.inaction_ok
is '1'b if the procedure that raised the condition has determined that it makes sense for an on-unit for this condition to return without taking any corrective action. If '0'b, the on-unit must take some corrective action before

returning, or else continued computation may be undefined. Cflags.inaction_ok will never be '1'b unless cflags.return_ok is '1'b as well. No user program should change the state of this or any other member of cfh.cflags.

cflags.specifier
if '1'b, indicates that this condition is a PL/I I/O (PLIO) condition that requires a specifier pointer as well as a condition name to completely identify it. This specifier is usually a pointer to a PLIO file control block. The specifier must be the first member of the info structure.

cflags.mbz
is reserved for future expansion and must be '0'b.

version
identifies the version number (and hence the format) of this structure, and will currently always be 1.

cond_name_ptr
is a pointer to the name (char(32) varying) of the condition because of which the on-unit is being invoked.

ms_ptr
is a pointer to a structure which defines the state of the CPU at the time the condition occurred. In the case of hardware faults, ms_ptr will point to a Standard Fault Frame Header (ffh). In the case of software-initiated conditions, ms_ptr will point to a cfh. The two cases can be distinguished by the value of ms_ptr -> cfh.flags.fault_fr: if '00'b, the software case obtains; otherwise, the hardware case obtains.

info_ptr
is a pointer to an arbitrary structure containing auxiliary information about the condition. If null, no information is available. This pointer is copied directly from the corresponding argument to SIGNL$. If cflags.specifier is '1'b, the format of this structure is partially constrained as described above.

ms_len
is the length in words of the structure pointed to by ms_ptr.

info_len
is the length in words of the structure pointed to by info_ptr.

saved_cleanup_pb
is valid only if flags.cleanup_done is '1'b, and if valid is the former value of cfh.ret_pb (which has been overwritten by the nonlocal goto processor).

Note

Any procedure attempting to interpret the data contained in a cfh structure should be aware that, in the case of a crawlout, cfh.ms_ptr describes the machine state at the time the condition was generated. The stack history pertaining to that machine state has been lost as a result of the crawlout.

The machine state extant at the time the inner ring was entered is available, and is pointed to by cfh.ret_sb. This machine state will be a cfh or an ffh according to whether the inner ring was entered via a procedure call (cfh) or a fault (ffh). The value of cfh.ret_sb -> cfh.flags.fault_fr can be used to distinguish these cases.

In the case where a crawlout has not occurred, cfh.ms_ptr points to the proper machine state, and no assumptions can be made concerning cfh.ret_sb.

The Extended Stack Frame Header

Any procedure (or begin block) that desires to create one or more on-units must reserve space in its stack frame header for an extension that contains descriptive information about those on-units. This space is allocated simply by including in such procedures, the proper declaration for the subroutine MKONU$.

The format of the stack frame header (with extension) is:

```
dcl    1 sfh based,  /* stack frame header */
           2 flags,
             3 backup_inh bit(1),
             3 cond_fr bit(1),
             3 cleanup_done bit(1),
             3 efh_present bit(1),
             3 user_proc bit(1),
             3 mbz bit(9),
             3 fault_fr bit(2),
           2 root,
             3 mbz bit(4),
             3 seg_no bit(12),
           2 ret_pb ptr,
           2 ret_sb ptr,
           2 ret_lb ptr options (short),
           2 ret_keys bit(16) aligned,
           2 after_pcl fixed bin,
           2 hdr_reserved(8) fixed bin,
           2 owner_ptr ptr options (short),
           2 tempsc(8) fixed bin,
           2 onunit_ptr ptr options (short),
           2 cleanup_onunit_ptr ptr options (short),
           2 next_efh ptr options (short);
```

flags.backup_inh
is examined only if this stack frame is the "crawlout frame" on an inner ring stack, and a crawlout is taking place. If '1'b, it indicates that sfh.ret_pb is to be copied to the outer ring as-is, so that the operation being aborted by the crawlout will not be retried. If '0'b, sfh.ret_pb will be set to point at the pcl instruction so that the inner ring call may be retried.

flags.cond_fr
will be '0'b unless the frame is a condition frame (and is hence described by the structure "cfh").

flags.cleanup_done
is '1'b if the nonlocal goto processor has "cleaned up" this frame by invoking its CLEANUP$ on-unit if any, and resetting its sfh.ret_pb to point to a special code sequence to accomplish the unwinding of this stack frame. When '1'b, the former value of sfh.ret_pb may be found in sfh.tempsc(7:8) provided sfh.flags.efh_present is set.

flags.efh_present
is '1'b if the extension portion of this frame header has been validly initialized. In the present implementation, this implies that at least one call to MKONU$ has been made, since MKONU$ is responsible for performing the initialization. If '0'b, members of this structure below marked (EFH) are not valid and may be used by the procedure for automatic storage.

flags.user_proc
is '1'b if this stack frame belongs to a "non-support" procedure; else is '0'b. If flags.user_proc is '1'b, sfh.owner_ptr is guaranteed to be valid, and to point to an ECB which is followed by the name of the entrypoint.

flags.mbz
is reserved and will be '0'b.

flags.fault_fr
is '00'b if this frame was created by a regular procedure call; or '10' if this frame is a fault frame (ffh) with valid saved registers; or '01'b if this frame is a fault frame (ffh) in which the registers have not yet been saved.

root.mbz
is reserved and must be '0'b.

root.seg_no
is the hardware-defined segment number of the stack root of the stack of which this frame is a member.

ret_pb
: points to the next instruction to be executed upon return from this procedure.

ret_sb
: contains the stack base belonging to the caller of this procedure, and hence also points to the immediate predecessor of this stack frame.

ret_lb
: contains the linkage base belonging to the caller of this procedure.

ret_keys
: contains the hardware-defined keys register belonging to the caller of this procedure.

after_pcl
: is a value such that the pcl instruction points to two words beyond the procedure call (PCL) instruction that invoked this procedure.

hdr_reserved(EFH)
: is reserved for future expansion of the hardware-defined PCL stack frame header.

owner_ptr(EFH)
: points to the Entry Control Block (ECB) of the procedure that owns this stack frame. This member must be initialized by the called procedure itself, as the PCL instruction does not do it.

tempsc_(EFH)
: is a fixed-position block of eight words to be used as temporary storage by procedures called by this procedure that have a "shortcall" invocation sequence and hence have no stack frame of their own.

onunit_ptr(EFH)
: points to the start of a chain of on-unit descriptor blocks for this activation. If onunit_ptr is null, this activation has no onunit blocks, except possibly for the condition CLEANUP$ as described below.

cleanup_onunit_ptr(EFH)
: If nonnull, this activation has an on-unit for the special condition CLEANUP$, and cleanup_onunit_ptr points to the ECB for that on-unit procedure (it does <u>not</u> point to an on-unit descriptor block).

next_efh(EFH)
points to the first on a chain of additional stack frame "header" blocks, so that these do not have to be allocated at the beginning of the stack frame. Presently, next_efh will always be null.

The Standard Fault Frame Header

Whenever a hardware fault occurs, the Fault Interceptor Module (FIM) is expected to push a stack frame with the standard format shown below.

The standard fault frame header structure is:

```
dcl 1 ffh based,  /* standard fault frame header */
      2 flags,
        3 backup_inh bit(1),
        3 cond_fr bit(1),
        3 cleanup_done bit(1),
        3 efh_present bit(1),
        3 user_proc bit(1),
        3 mbz bit(9),
        3 fault_fr bit(2),
      2 root,
        3 mbz bit(4),
        3 seg_no bit(12),
      2 ret_pb ptr,
      2 ret_sb ptr,
      2 ret_lb ptr options (short),
      2 ret_keys bit(16) aligned,
      2 fault_type fixed bin,
      2 fault_code fixed bin,
      2 fault_addr ptr options (short),
      2 hdr_reserved(7) fixed bin,
      2 regs,
        3 save_mask bit(16) aligned,
        3 fac_1(2) fixed bin(31),
        3 fac_0(2) fixed bin(31),
        3 genr(0:7) fixed bin(31),
        3 xb_reg ptr options (short),
      2 saved_cleanup_pb ptr options (short),
      2 pad fixed bin;
```

flags.backup_inh
will be ignored by the Condition Mechanism for fault frames.

flags.cond_fr
will be '0'b in a fault frame.

flags.cleanup_done
: is set to '1'b by the stack unwinder when it has "cleaned up" this fault frame. The old value of ffh.ret_pb has been placed in ffh.saved_cleanup_pb, provided flags.fault_fr is '10'b.

flags.efh_present
: will be '0'b in a fault frame, implying that FIM's may not make on-units.

flags.user_proc
: will always be '0'b in a fault frame.

flags.mbz
: is reserved and will be '0'b.

flags.fault_fr
: will be '10'b if this frame is indeed a standard format ffh and the registers have been validly saved in ffh.regs; else will be '01'b.

root.seg_no
: is the hardware-define stack root segment number.

ret_pb
: points to the next instruction to be executed following a return from the fault. This will frequently also be the instruction that caused the fault (the case for those faults defined by the CPU reference manual as "backing up" the program counter). If flags.cleanup_done is '1'b, ret_pb will point to a special "unwind" code sequence, and its former value will have been saved if possible in ffh.saved_cleanup_pb.

ret_sb
: contains the value of the SB register at the time of the fault, and hence will usually point to the predecessor of this stack frame.

ret_lb
: contains the value of the LB register at the time of the fault.

ret_keys
: contains the value of the KEYS register at the time of the fault. This can be used to determine in what addressing mode the fault was taken.

fault_type
: is set by each FIM to the offset in the fault table corresponding to the fault that occurred (e.g. a Process Fault results in a fault_type of '04'b3). This datum cannot be guaranteed valid, as it is not set indivisibly with the hardware-defined header information. Since FIM's usually set fault_type just after saving the registers, it is very unlikely for fault_type to be invalid.

fault_code
is the hardware-defined fault code produced by the fault that was taken.

fault_addr
is the hardware-defined fault address produced by the fault that was taken.

hdr_reserved
is reserved for future expansion of the hardware-defined stack header.

regs
is valid if flags.fault_fr is '1Ø'b, and if valid contains the saved machine registers at the time of the fault, in the format produced by the RSAV instruction.

saved_cleanup_pb
is valid only if flags.fault_fr is '1Ø'b and flags.cleanup_done is '1'b, and if valid contains the value that was in ret_pb before the latter was overwritten by the stack unwinder.

pad
exists only to make the size of this structure an even number of words.

The On-Unit Descriptor Block

Each on-unit created by an activation is described to the condition mechanism by a descriptor block (except for the special condition CLEANUP$, which has no descriptor). These descriptor blocks are threaded together in a simple linked list, the head of which is pointed to by sfh.onunit_ptr. The format of an on-unit descriptor is:

```
dcl    1 onub based,  /* standard onunit block */
           2 ecb_ptr ptr options (short),
           2 next_ptr ptr options (short),
           2 flags,
             3 not_reverted bit(1),
             3 is_proc bit(1),
             3 specify bit(1),
             3 snap bit(1),
             3 mbz bit(12),
           2 pad fixed bin,
           2 cond_name_ptr ptr options (short),
           2 specifier_ptr options (short);
```

ecb_ptr
: points to the Entry Control Block (ECB) which represents the procedure or begin block to be invoked when this on-unit is selected for invocation.

next_ptr
: points to the next on-unit descriptor on the chain for this activation, or else is null if at the end of the list.

flags.not_reverted
: is '1'b if this on-unit is still valid and has not been reverted, and is '0'b if the on-unit has been reverted and is to be ignored by the condition raising mechanism.

flags.is_proc
: is '1'b if this on-unit was made via a call to the primitive MKONU$, and '0'b if it was made via the PL/I <on statement>.

flags.specify
: is '1'b if the condition name does not fully identify which condition this on-unit block is to handle: onub.specifier is a further qualifier in this case.

flags.snap
: is '1'b if the <snap option> was specified in the PL/I <on statement> that created this on-unit; otherwise it is '0'b.

flags.mbz
: is reserved and must be '0'b.

pad
: is reserved and must be 0.

cond_name_ptr
: is a pointer to a varying character string containing the condition name for which this on-unit is a handler. This name may be an incomplete specification if onub.flags.specify is '1'b.

specifier
: is valid only if onub.flags.specify is '1'b, and if valid qualifies the condition name that is pointed to by onub.cond_name_ptr. The primary use of onub.specifier is for PL/I I/O conditions, in which the specification of the condition requires both a name and a file descriptor pointer.

APPENDIX A

FORTRAN INTERNAL SUBROUTINES

INTERNAL SUBROUTINES

The following subroutines are used internally by the FORTRAN compiler. They may be of some value to the user and are briefly described. For calling sequence and further information, refer to the compiler or library source listings.

Table A-1. Subroutines Internal to FORTRAN

Subroutine	Function
F$TR	Perform the function of the FORTRAN TRACE routine.
F$RN	Read with no alternate returns.
F$RNX	Read with ERR= and END= alternate returns.
F$WN	Write with no alternate returns
F$WNX	Write with ERR= alternate return.
F$DN	Close (END-FILE) logical device specified.
F$FN	Provide backspace function to FORTRAN run-time programs.
F$BN	Rewind logical device specified.
F$CB	End of READ/WRITE statement.
F$A1	Input/output 16-bit integer.
F$A2	Input/output single-precision floating-point.
F$A3	Input/output logical.
F$A5	Input/output complex.
F$A6	Input/output double-precision floating-point.
F$A7	Input/output long integer.
F$BKSP	Backspace statement processor.
F$CG	FORTRAN computed GOTO processor.

Table A-1. Subroutines Internal to FORTRAN (continued)

Subroutine	Function
F$CLOS	Close statement processor.
F$OPEN	Open statement processor.
F$RA	Read ASCII, no alternate returns.
F$INQU	Inquire by unit statement processor.
F$INQF	Inquire by file statement processor.
F$PAUS	Pause statement processor.
F$RB	Read BINARY, no alternate returns.
F$RAX	Read ASCII, with ERR= and END= alternate returns.
F$RBX	Read BINARY with ERR= and END= alernate returns.
F$RX	COMMON read handler.
F$STOP	Stop statement processor.
F$WA	Write ASCII, no alternate returns.
F$WB	Write BINARY, no alternate returns.
F$WAX	Write ASCII with ERR= and END= alternate returns.
F$WBX	Write BINARY, with ERR= and END= alternate returns.
F$WX	COMMON write handler.
F$EN	Encode statement processor.
F$END	Endfile statement processor.
F$DE	Decode statement processor.
F$DEX	Decode statement processor with ERR=.
F$IOFTN	Read and write records in manner compatible with F$IO
F$IO77	Read and write variable-length records in default case of F$IO.

Table A-1. Subroutines Internal to FORTRAN (continued)

Subroutine	Function
F$IFW	Initialize formatted write.
F$IFR	Initialize formatted read.
F$INR	Initialize namelist read.
F$IBW	Initialize unformatted write.
F$IBR	Initialize unformatted read.
F$ILDR	Initialize list-directed read.
F$ILDW	Initialize list-directed write.
F$IOBF	F$IO buffer definition (up to 128 words, for R-mode and non-shared V-mode; up to 16K-1 words in shared V-mode library).
F$REW	Rewind statement processor.
F$RTE	FORTRAN RETURN statement processor.
F$AT	FORTRAN R-mode argument transfer subroutine.
F$ATI	FORTRAN argument transfer subroutine for PROTECTED subroutine.

INTRINSIC FUNCTIONS

The following subroutines are the FORTRAN library intrinsic function handlers:

Subroutine	Function
F$LT	Left truncate
F$RT	Right truncate
F$LS	Left shift
F$RS	Right shift
F$SH	General shift
F$OR	Inclusive OR

FLOATING POINT EXCEPTIONS

The FLEX (and F$FLEX) subruotines are invoked by the compiler or system. This subroutine is the floating point exception interrupt processor. It determines the exception type, which may be:

Exponent overflow/underflow

Divide by zero

Store exception

Real-integer exception

A message is returned as follows:

Exponent Overflow	SE
Exponent Underflow	DE
Divide by Zero	DZ
Store Exception	SE
Real-Integer Exception	RI

For further information on floating point exception (FLEX), refer to the System Architecture Reference Guide (PDR3Ø6Ø).

APPENDIX B

INDICATION AND CONTROL SUBROUTINES

OVERVIEW

These subroutines return a message or an error indicator value in AC5 or set a value depending on some machine condition.

They include:

OVERFL
SLITE SLITET SSWTCH
DISPLY

These subroutines are not currently available in V-mode under PRIMOS.

SUBROUTINE DESCRIPTIONS

DISPLY

DISPLY updates the sense light settings according to argument A1. The bit values of A1 (1=ON, Ø=OFF) correspond to switch/light settings which are displayed on the computer control panel.

```
CALL DISPLY (A1)
```

OVERFL

Argument A1 in location AC5 is given a value 1 if entry into F$ER was made; otherwise it is set to 2. F$ER is left in the no error condition. OVERFL is called to check if an overflow condition has occurred.

```
CALL OVERFL (A1)
```

SLITE

Sets the sense light specified in argument A1 ON or sets all sense lights OFF. If A1=Ø, all sense lights are reset OFF.

```
CALL SLITE (A1)
CALL SLITE (Ø)
```

SLITET

SLITET tests the setting of a sense light specified by the argument A1. The result of this test (1 for ON, 2 for OFF) is in the location specified by the argument R.

```
CALL SLITET (A1,R)
```

SSWTCH

SSWTCH tests the setting of a sense Switch specified by the argument A1. The result of this test (1=SET, 2=RESET) is stored in the location specified in argument R.

```
CALL SSWTCH (A1,R)
```

APPENDIX C

SVC INFORMATION

SVC's CALLED BY PRIMOS SUBROUTINES

This appendix defines SVC's called by PRIMOS subroutines.

Note

* => Also Direct Entrance Call.

```
*1500  ATCH$$ (ufdnam,namlen,ldisk,passwd,(key code))
 1400  ATTAC$ (ufdnam,namlen,ldisk,passwd,(key,loc (code)))
 0100  ATTACH (ufdnam,ldisk,paswd,(key,altrtn))
*0507  BREAK$ (offon)
*0601  ClIN   (char)
 0602  CMREAD (char)
*1515  CNAM$$ (oldnam,oldlen,newnam,newlen,code)
 0113  CNAME  (oldnam,newnam,altrtn)
 1415  CNAME$ (oldnam,oldlen,newnam,newlen,loc(code))
*0604  CNIN$  (buff,charcnt,statv(3))
*0600  COMANL
*1516  COMI$$ (filnam,namlen,unit,code)
 1416  COMIN$ (filnam,namlen,unit,loc(code))
 0603  COMINP (filnam,unit,(altrtn))
*1523  COMO$$ (key,filnam,namlen,xxxxxx,code)
 0401  CONECT (tgtnam,tgtusr,lun,data,statv,lintyp)
*1501  CREA$$ (ufdnam,namlen,opass,npass,code)
 1401  CREAT$ (ufdnam,namlen,opass,npass,loc(code))
 0506  D$INIT (pdev)
 0410  DISCON (lun,data,statv)
*0705  DUPLX$ (key)
*1524  ERKL$$ (key,erasec,killc,code)
*1402  ERRPR$ (key,code),text,txtlen,name,namlen)
 0106  ERRRTN (altrtn,name,msg,msglen)
 0114  ERRSET (altval,altrtn,name,msglen)
*0105  EXIT
 0400  FAMSVC (a1,a2,a3,a4,a5,a6,altrtn)
*0115  FORCEW (key,unit)
 0402  GETCON (target,user,data,statv)
 0110  GETERR (buff,nw)
 0112  GINFO  (buff,nw)
*1504  GPAS$$ (ufdnam,namlen,opass,npass,code)
 1404  GPASS$ (ufdnam,namlen,opass,npass,code)
 0412  NETLNK (statv)
 0406  NETWAT
 0407  NTSTAT (key,p1,p2,array)
 0111  PRERR
*1506  PRWF$$ (key,Funit,loc(bf),bflen,pos32,rnw,code)
 0300  PRWFIL (key,unit,loc(buff),n,pos,altrtn)
```

```
 1406  PRWFL$ (key,unit,loc(buff),nw,pos,rnw,loc(code))
*1507  RDEN$$ (key,funit,bf,bfln,rnw,nam32,namln,code)
 1407  RDENT$ (key,unit,buff,buflen,Rnw,name32,namlen,loc(code))
 0202  RDLIN  (unit,line,nw,altrtn)
*1525  RDLIN$ (unit,line,nw,code)
*1517  RDTK$$ (key,info(8),buff,buflen,code)
 1417  RDTKN$ (key,info(8),buff,buflen,loc(code))
 0404  RECEIV (lun,loc(buff),nw,statv)
*0505  RECYCL
*1520  REST$$ (rvec,name,namlen,code)
 1420  RESTO$ (rvec,name,namlen,loc(code))
 0103  RESTOR (rvec,name,altrtn)
*1521  RESU$$ (name,namlen)
 1421  RESUM$ (name,namlen)
 0104  RESUME (name)
 0403  RJCON  (target,user,statv,numtyp)
 0500  RREC   (loc(buff),buflen,n,ra,pdev,(altrtn))
 0516  RRECL  (loc(buff),buflen,n,ra32,pdev,(altrtn))
*1510  SATR$$ (key,name,namlen,array,code)
 1410  SATTR$ (key,name,namlen,array,loc(code))
 0102  SAVE   (rvec,name)
 1422  SAVE$  (rvec,name,namlen,loc(code))
*1522  SAVE$$ (rvec,name,namlen,code)
 1411  SEARC$ (key,name,namlen,unit,type,loc(code))
 0101  SEARCH (key,name,unit,(altrtn))
 1414  SEGDR$ (key,unit,entrya,entryb,loc(code))
*1512  SGDR$$ (key,funit,entrya,entryb,code)
* --   SEM$DR (semnum,code)
* --   SEM$NF (semnum,code)
* --   SEM$TN (semnum,int32,int32,code)
* --   SEM$TS (semnum,code)  (int fcn)
* --   SEM$WT (semnum,code)
* --   SLEEP$ (int32)
*1513  SPAS$$ (opass,npass,loc(code))
 1413  SPASS$ (key,name,namlen,unit,type,code)
*1511  SRCH$$ (key,name,namlen,unit,type,code)
*0513  T$AMLC (line,loc(buff),nw,inst.statv)
*0512  T$CMPC (unit,loc(buff),nw,inst,statv)
*0511  T$LMPC (unit,loc(buff),nw,inst,statv)
*0515  T$PMPC (unit,loc(buff),nw,inst,statv)
*0510  T$MT   (unit,loc(buff),nw,inst,statv)
*0514  T$VG   (unit,loc(buff),nw,inst,statv)
 1001  T$SLC  (key,line,loc(buff),nw)
*0502  TIMDAT (buff,buflen)
*0702  TNOU   (msg,charcnt)
*0703  TNOUA  (msg,charcnt)
 0405  TRNMIT (lun,loc(buff),cnt,statv)
 0411  UNLINK
 0501  WREC   (loc(buff),buflen,n,ra,pev,(altrtn))
 0517  WRECL  (loc(buff),buflen,n,ra32,pdev,(altrtn))
 0203  WTLIN  (unit,line,nw,(altrtn))
*1526  WTLIN$ (unit,line,nw,code)
```

SVC INTERFACE FOR I-O CALLS

The I/O subroutines described in Section 15 interface with the operating system by means of supervisor call instructions (SVC's). This Appendix describes these interfaces.

SVC INTERFACE CONSIDERATIONS

Disk

The disk interfaces with virtual memory through a supervisor call (SVC) instruction to perform a READ or WRITE operation on a single physical record of a physical disk. The disk must be assigned to the terminal by the ASSIGN command. Refer to RREC and WREC in Section 6. For information about the SVC instruction, refer to the Systems Reference Manual and the PMA User Guide.

Magnetic Tape

Input/Output operations for magnetic tape are accomplished by PRIMOS III through SVC calls. Refer to T$MT in the Section 15.

MPC Line Printer

Output to the parallel interface line printer is accomplished through SVC calls. Refer to T$LMPC in the Section 15.

MPC Card Reader

Input from the parallel interface card reader is controlled through SVC calls. Refer to T$CMPC in the Section 15.

Table C-1 is a list of SVC codes used by PRIMOS III (SVC codes are generally not applicable to PRIMOS users).

Table C-1. SVC Numbers Used by PRIMOS III.

SVC Number	Associated Call	
100	ATTACH	(ufdnam, ldev, passwd, key, altrtn)
1	SEARCH	(key, name, unit, altrtn)
2	SAVE	(rvec, name)
3	RESTOR	(rvec, name, altrtn)
4	RESUME	(name)
5	EXIT	
6	ERRTN	(altrtn, a1, a2, a3)
7	UPDATE	(1,0)
110	GETERR	(buff, nw)
1	PRERR	
2	GINFO	(abuff, nw)
3	CNAME	(oldnam, newnam, altrtn)
4	ERRSET	(altval, altrtn, a1, a2, a3)
5	FORCEW	(key, unit)
202	RDLIN	(unit, line, nw, altrtn)
3	WTLIN	(unit, line, nw, altrtn)
300	PRWFIL	(key, unit, LOC(buff), nw, posv, altrtn)
500	RREC	(pbav, nwv, nchn, ra, pdev, altrtn)
1	WREC	(pbav, nwv, nchn, ra, pdev, altrtn)
2	TIMDAT	(buff, nw)
3	—	reserved
4	--	reserved
5	RECYCL	
6	D$INIT	(pdev)
7	BREAK$	(onoff)
510	T$MT	(unit, LOC(buff), nw, inst, statv)
1	T$LMPC	(unit, LOC(buff), nw, inst, statv)
2	T$CMPC	(unit, LOC(buff), nw, inst, statv)
3	T$AMLC	(line, ba, charent, key, statv, altrtn)
4	T$VG	(unit, ba, nw, inst, statv)
600	COMANL	
1	CLIN	(char)
2	CMREAD	(buff)
3	COMINP	(name, unit, altrtn)
4	CNIN$	(buff, charent)
700	T1IN	(char)
1	T1OU	(char)
2	TNOU	(msg, cnt)
3	TNOUA	(msg, cnt)
4	TOOCT	(num)
5	DUPLX$	(argument)

1ØØØ	T$MT	See 51Ø
1	T$SLC	(key, lin LOC(buff), nw)
11ØØ	T$LMPC	See 511
12ØØ	T$CMPC	See 512

OPERATING SYSTEM RESPONSE TO SVC

The operating system response to supervisor calls includes a "return to sender" capability. The format is an SVC instruction followed by a word encoded as follows:

Bits	Meaning
1	Use interlude routine
2	Return to sender
3-4	Zero
5-1Ø	SVC class
11-16	SVC sub-class

When bit 1 is set, the operating system assumes the location preceding the SVC is a subroutine entry point and looks for the arguments back through that entry point.

When bit 2 is set, the operating system either performs the requested function or, if the class and sub-class are not recognized, returns to the caller at the location following the SVC code word.

The four legal syntaxes are:

1.

```
      .
      .
      .
    SVC
    OCT    ØØxxyy
    DAC
    DAC
      .
      .
      .
    OCT    Ø
```

2.

```
    Ent  DAC  **
         SVC
         OCT  1Øxxyy
```

3.

```
      .
      .
      .
    SVC
    OCT    Ø4xxyy
    (return-to-sender location)
    DAC
    DAC
      .
      .
      .
    OCT    Ø
```

4.

```
    Ent  DAC  **
         SVC
         OCT  14xxyy
         (return-to-sender location)
           .
           .
           .
```

where xx = 6 bit class
yy = 6 bit sub-class

The following classes are currently assigned:

Ø	RTOS
1	File system miscellaneous
2	Sequential File I/O
3	Direct File I/O
4	-
5	DOSVM only; never reflected
6	Command input/output
7	Typers
1Ø	Mag Tape
11	Line Printer
12	Card Reader/Punch
13	SMLC
77	Reserved for Customer Usage

APPENDIX D

KEYS (SYSCOM KEYS.F)

This appendix summarizes the keys associated with PRIMOS subroutine calls.

KEYS (SYSCOM>KEYS.F)

```
C SYSCOM>KEYS.F       MNEMONIC KEYS FOR FILE SYSTEM (FTN)      09/29/78
      NOLIST
C
C     TABSET 6 11 28 69
C
      INTEGER*2 K$READ,K$WRIT,K$POSN,K$TRNC,K$RPOS,K$PRER,K$PREA,
     X    K$POSR,K$POSA,K$CONV,K$RDWR,K$CLOS,K$DELE,K$EXST,K$GETU,
     X    K$IUFD,K$ISEG,K$CACC,K$NSAM,K$NDAM,K$NSGS,K$NSGD,K$CURR,
     X    K$IMFD,K$ICUR,K$SETC,K$SETH,K$ALLD,K$SPOS,K$GOND,K$MSIZ,
     X    K$GPOS,K$UPOS,K$NAME,K$FRCW,
     X    K$PROT,K$DTIM,K$DMPB,K$RWLK,K$NRTN,K$SRTN,K$IRTN,K$HOME,
     X    K$MVNT,K$RSUB,K$FULL,K$FREE,K$CPLM,K$LGLM,
     X    K$UNIT,K$CURA,K$HOMA
C
      PARAMETER
     X
     X /*****************************************************************/
     X /*                                                               */
     X /*                                                               */
     X /*      KEY DEFINITIONS                                          */
     X /*                                                               */
     X /*                                                               */
     X /********************** PRWF$$ **********************             */
     X /*                ****** RWKEY  ******                           */
     X    K$READ = :1,     /* READ                                     */
     X    K$WRIT = :2,     /* WRITE                                    */
     X    K$POSN = :3,     /* POSITION ONLY                            */
     X    K$TRNC = :4,     /* TRUNCATE                                 */
     X    K$RPOS = :5,     /* READ CURRENT POSITION                    */
     X /*               ****** POSKEY ******                           */
     X    K$PRER = :0,     /* PRE-POSITION RELATIVE                    */
     X    K$PREA = :10,    /* PRE-POSITION ABSOLUTE                    */
     X    K$POSR = :20,    /* POST-POSITION RELATIVE                   */
     X    K$POSA = :30,    /* POST-POSITION ABSOLUTE                   */
     X /*               ****** MODE   ******                           */
     X    K$CONV = :400,   /* CONVENIENT NUMBER OF WORDS               */
     X    K$FRCW = :40000, /* FORCED WRITE TO DISK                     */
     X /*                                                              */
     X /********************** SRCH$$ **********************            */
     X /*              ****** ACTION ******                            */
```

```
X /* K$READ = :1,     /* OPEN FOR READ                                */
X /* K$WRIT = :2,     /* OPEN FOR WRITE                               */
X    K$RDWR = :3,     /* OPEN FOR READING AND WRITING                 */
X    K$CLOS = :4,     /* CLOSE FILE UNIT                              */
X    K$DELE = :5,     /* DELETE FILE                                  */
X    K$EXST = :6,     /* CHECK FILE'S EXISTENCE                       */
X    K$GETU = :40000, /* SYSTEM RETURNS UNIT NUMBER                   */
X /*              ****** REF    ******                                */
X    K$IUFD = :0,     /* FILE ENTRY IS IN UFD                         */
X    K$ISEG = :100,   /* FILE ENTRY IS IN SEGMENT DIRECTORY           */
X    K$CACC = :1000,  /* CHANGE ACCESS                                */
X /*              ****** NEWFIL ******                                */
X    K$NSAM = :0,     /* NEW SAM FILE                                 */
X    K$NDAM = :2000,  /* NEW DAM FILE                                 */
X    K$NSGS = :4000,  /* NEW SAM SEGMENT DIRECTORY                    */
X    K$NSGD = :6000,  /* NEW DAM SEGMENT DIRECTORY                    */
X    K$CURR = :177777,/* CURRENTLY ATTACHED UFD                       */
X /*                                                                   */
X /********************* ATCH$$ *********************                 */
X /*               ****** KEY    ******                               */
X    K$IMFD = :0,     /* UFD IS IN MFD                                */
X    K$ICUR = :2,     /* UFD IS IN CURRENT UFD                        */
X /*               ****** KEYMOD ******                               */
X    K$SETC = :0,     /* SET CURRENT UFD (DO NOT SET HOME)            */
X    K$SETH = :1,     /* SET HOME UFD (AS WELL AS CURRENT)            */
X /*               ****** NAME   ******                               */
X    K$HOME = :0,     /* RETURN TO HOME UFD (KEY=K$IMFD)              */
X /*               ****** LDISK  ******                               */
X    K$ALLD = :100000,/* SEARCH ALL DISKS                             */
X /* K$CURR = :177777,/* SEARCH MFD OF CURRENT DISK                   */
X /*                                                                   */
X /********************* SGDR$$ *********************                 */
X /*               ****** KEY    ******                               */
X    K$SPOS = :1,     /* POSITION TO ENTRY NUMBER IN SEGDIR           */
X    K$GOND = :2,     /* POSITION TO END OF SEGDIR                    */
X    K$GPOS = :3,     /* RETURN CURRENT ENTRY NUMBER                  */
X    K$MSIZ = :4,     /* MAKE SEGDIR GIVEN NR OF ENTRIES              */
X    K$MVNT = :5,     /* MOVE FILE ENTRY TO DIFFERENT POSITION        */
X    K$FULL = :6,     /* POSITION TO NEXT NON-EMPTY ENTRY             */
X    K$FREE = :7,     /* POSITION TO NEXT FREE ENTRY                  */
X /*                                                                   */
X /********************* RDEN$$ *********************                 */
X /*               ****** KEY    ******                               */
X /* K$READ = :1,     /* READ NEXT ENTRY                              */
X    K$RSUB = :2,     /* READ NEXT SUB-ENTRY                          */
X /* K$GPOS = :3,     /* RETURN CURRENT POSITION IN UFD               */
X    K$UPOS = :4,     /* POSITION IN UFD                              */
X    K$NAME = :5,     /* READ ENTRY SPECIFIED BY NAME                 */
X /*                                                                   */
X /********************* SATR$$ *********************                 */
X /*               ****** KEY    ******                               */
X    K$PROT = :1,     /* SET PROTECTION                               */
X    K$DTIM = :2,     /* SET DATE/TIME MODIFIED                       */
```

```
X    K$DMPB = :3,     /* SET DUMPED BIT                          */
X    K$RWLK = :4,     /* SET PER FILE READ/WRITE LOCK            */
X /*                                                             */
X /********************* ERRPR$ ********************             */
X /*                ****** KEY    ******                         */
X    K$NRTN = :0,     /* NEVER RETURN TO USER                    */
X    K$SRTN = :1,     /* RETURN AFTER START COMMAND              */
X    K$IRTN = :2,     /* IMMEDIATE RETURN TO USER                */
X /*                                                             */
X /********************* LIMIT$ ************************          */
X /*                ****** KEY    ******                         */
X /* K$READ = :0,     /* RETURNS INFORMATION                     */
X /* K$WRIT = :1,     /* SETS INFORMATION                        */
X /*                ****** SUBKEY ******                         */
X    K$CPLM = :400,   /* CPU TIME IN SECONDS                     */
X    K$LGLM = :1000,  /* LOGIN TIME IN MINUTES                   */
X /*                                                             */
X /*                                                             */
X /********************* GPATH$ ********************************/
X /*                ****** KEY    ******                         */
X    K$UNIT = :1,     /* PATHNAME OF UNIT RETURNED               */
X    K$CURA = :2,     /* PATHNAME OF CURRENT ATTACH POINT        */
X    K$HOMA = :3      /* PATHNAME OF HOME ATTACH POINT           */
X /*                                                             */
X /*************************************************************/
 LIST
```

APPENDIX E

INTERNAL FILE FORMATS

The internal formats of all disk records for both the old and new file management system are described below. They have been collected for ease in noting the changes that have been made. User programs will normally have no need to refer to the internal file system formats. Where possible, field names are the same as those used by the internal file system routines. Numbers preceeded by a ':' are octal, otherwise they are decimal.

DSKRAT FORMATS

DSKRAT Format -- Old Partitions

Ø	5	NUMBER OF WORDS IN DSKRAT HEADER = 5
1	RECSIZ	DISK RECORD SIZE (448 or 1Ø4Ø)
2	NMRECS	NUMBER OF RECORDS IN PARTITION
3	UNUSED	UNUSED
4	NHEADS	NUMBER OF HEADS IN PARTITION
5	DATA	START OF DKSRAT DATA (ONE BIT/RECORD)
	....	

DSKRAT Format -- New Partitions

Ø	8	NUMBER WORDS IN HEADER = 8
1	RECSIZ	RECORD SIZE
2	NMRECS	NUMBER OF RECORDS IN PARTITON (TWO WORDS)
4	NHEADS	NUMBER OF HEADS IN PARTITION
5	RESERVED	RESERVED
6	RESERVED	RESERVED
7	RESERVED	RESERVED
8	DATA	START OF DSKRAT DATA (ONE BIT/RECORD)
	....	

RECORD HEADER FORMATS

Record header formats are the same for new and old partitions. The format of a record header is a function of the physical record size.

Record Header Format -- 448-Word Records

Ø	REKCRA	RECORD ADDRESS (OF THIS RECORD)
1	REKBRA	RA OF DIRECTORY ENTRY OR FIRST RECORD
2	REKFPT	RA OF NEXT SEQUENTIAL RECORD
3	REKBPT	RA OF PREVIOUS RECORD
4	REKCNT	NUMBER OF DATA WORDS IN FILE
5	REKTYP	TYPE OF THIS FILE
6	REKLVL	INDEX LEVEL FOR NEW PARTITION DAM FILES
7	RESERVED	RESERVED

Record Header Format -- 1040-Word Records

Ø	REKCRA	RECORD ADDRESS OF THIS RECORD (TWO WORDS)
2	REKBRA	BEGINNING RECORD ADDRESS (TWO WORDS)
4	REKCNT	NUMBER DATA WORDS IN THIS RECORD
5	REKTYP	TYPE OF THIS FILE
6	REKFPT	RA OF NEXT SEQUENTIAL RECORD (TWO WORDS)
8	REKBPT	RA OF PREVIOUS RECORD (TWO WORDS)
1Ø	REKLVL	INDEX LEVEL FOR NEW PARTITION DAM FILES
11		
	RESERVED	RESERVED (FIVE WORDS)
15		

Notes

1. Storage Modules have 1040-word records. All other disks have 448-word records.

2. The BRA of the first record in a file points to the beginning record address of the directory in which the file entry appears. In all other records, the BRA points to the first record of the file.

3. REKFPT contains the address of the next sequential record in the file or Ø if it is the last record in the file.

4. REKBPT contains the address of the previous record in sequence or Ø if it is the first record in the file.

5. REKTYP is valid only in the first record of a file. Legal values are:
   - Ø SAM File
   - 1 DAM File
   - 2 SAM Segment Directory
   - 3 DAM Segment Directory
   - 4 User File Directory (UFD)

6. If the file is the record zero bootstrap (BOOT) or the disk record availability table (DSKRAT or volume name) and the disk has a 1Ø4Ø record size (Storage Module), bit 1 (:1ØØØØØ) of REKTYP will be set.

7. DAM files on new partitions are organized somewhat differently from the above.

UFD HEADER AND ENTRY FORMATS

Old UFD Header Format

Ø	8	SIZE OF HEADER -- 8 WORDS
1	OPASSW	OWNER PASSWORD (THREE WORDS)
4	NPASSW	NON-OWNER PASSWORD (THREE WORDS)
7	RESERVED	RESERVED

New UFD Header Format

Ø	ECW	ECW (SEE NOTE 1 BELOW)
1	OPASSW	OWNER PASSWORD (THREE WORDS)
4	NPASSW	NON-OWNER PASSWORD (THREE WORDS)
7		
	RESERVED	RESERVED (SIXTEEN WORDS)
23		

Old UFD Entry Format

```
 Ø | BRA    |  BEGINNING RECORD ADDRESS
 1 | FILE   |  FILENAME (THREE WORDS)
   |        |
   | NAME   |
 4 | SPACES |  TWO BLANKS FOR NAME EXPANSION (RESERVED)
 5 | PROTEC |  PROTECTION (OWNER/NON-OWNER)
```

Note

In an old UFD, the high-order eight bits of PROTEC are the owner rights stored in complemented form (Ø=>owner has right). The low-order eight bits are non-owner protection, stored in true form (Ø=>no right). On creation, PROTEC=Ø. After a 'PROT 7 Ø', PROTEC=:174ØØØ.

New UFD Entry Format

```
 Ø | ECW    |  ENTRY CONTROL WORD (TYPE/LENGTH)
 1 | BRA    |  BEGINNING RECORD ADDRESS (TWO WORDS)
   |        |
 3 |RESERVED|  RESERVED (THREE WORDS)
   |        |
   |        |
 6 | PROTEC |  PROTECTION (OWNER/NON-OWNER)
 7 |RESERVED|  RESERVED FOR FUTURE USE
 8 | DATMOD |  DATE LAST MODIFIED (YYYYYYYMMMMDDDDD)
 9 | TIMMOD |  TIME LAST MODIFIED (SECONDS-SINCE-MIDNIGHT/4)
1Ø | FILTYP |  FILETYPE
11 | SCW    |  SUBENTRY CONTROL WORD FOR FILENAME
12 |F       |
   | I      |
   |   L    |
   |     E  |
   | ...    |  FILENAME (1-16 WORDS, BLANK PADDED)
   |N       |
   | A      |
   |   M    |
 N |     E  |
```

Notes

1. The Entry Control Word (ECW) consists of two eight-bit subfields. The top eight bits indicate the type of the following entry as follows:

Ø	Old UFD Header
1	New UFD Header
2	Vacant Entry
3	New UFD File Entry

The low-order eight bits give the size of the entry <u>including</u> the ECW itself.

2. The bits in PROTEC are stored in true form (Ø=> no right) for <u>both</u> owner and non-owner fields.

3. The file type field is as before (see Old Record Header Format) with following additional bits:

BIT	MEANING WHEN BIT IS ON
1	File has 16-word header (DSKRAT and BOOT only).
2	Change bit. Set by call to SATR$$, then reset
4	Special file (BOOT, DSKRAT, MFD, BADSPT).

4. The Subentry Control Word (SCW) consists of two eight-bit subfields. The top 8 bits are Ø, indicating subentry type Ø. The low-order 8 bits give the size of the subentry including the SCW itself.

5. UFD entries are reused by the file management system. Therefore, a new entry will not necessarily appear at the end of the UFD.

SEGMENT DIRECTORY FORMATS

<u>Old Segment Directory Format</u>

Ø	BRAØ	BRA OF FIRST ENTRY IN DIRECTORY
1	BRA1	BRA OF SECOND FILE
2	ØØØØ	NULL ENTRY
	....	
n	BRAn	BRA OF LAST FILE IN DIRECTORY

<u>New Segment Directory Format</u>

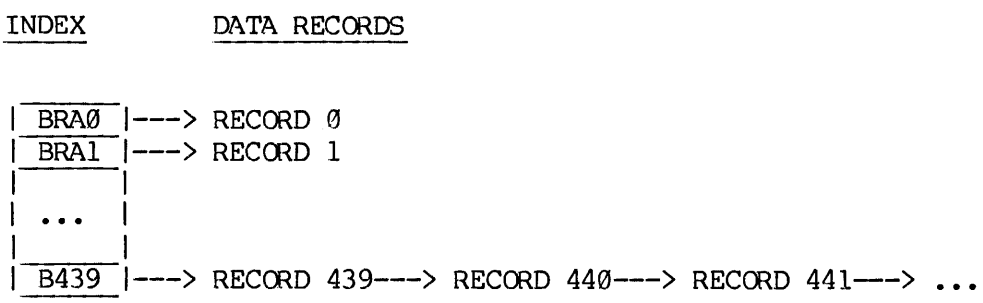
Ø	BRAØ	BRA OF FIRST FILE IN DIRECTORY (TWO WORDS)
2	BRA1	BRA OF SECOND FILE IN DIRECTORY (TWO WORDS)
	ØØØØ	NULL ENTRY (TWO WORDS)
	ØØØØ	
	....	
2n	BRAn	BRA OF LAST FILE IN DIRECTORY (TWO WORDS)

Note

The only difference between old and new directories is that each entry has been expanded to two words. A null entry in a new directory is a 32-bit Ø.

DAM FILE ORGANIZATION

In old-style DAM files, the first physical record of the file was reserved to be an index to the first 44Ø or 1Ø24 (depending on physical record size) records in the file. When this index was filled, however, access to subsequently added records became sequential. For example, in the file shown below, records Ø-439 can be accessed directly. Records 44Ø and above must be searched for sequentially starting with record 439.



The major difference between new and old DAM files is that the index is not limited to a single record, but can grow into a multi-level tree. (Also, since pointers are now two words each, each index record holds half the number of pointers in old index records.) An index can grow to any size, and any data record can be directly accessed. The following paragraphs explain how this multi-level index is built.

The handling of a DAM file on a new partition is identical to that on an old partition up to the point at which the index record is full and another record is to be added to the file. At this point the following actions take place.

1. Three new records are obtained from the file system. One of these records is to be the new data record, the other two are used to construct the second index level.

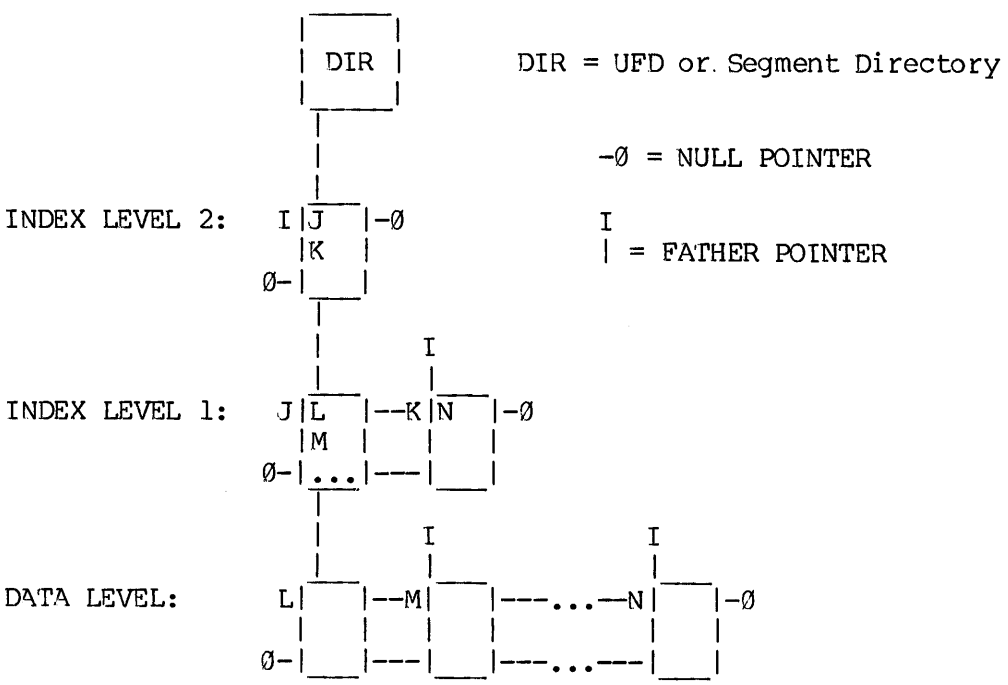
2. The index entries from the full index record are copied into one of the other new records. This record is to become the first index record of the new index level.

3. The old index record is reinitialized to contain two pointers to the two index records on the new level.

4. The other new index record is initialized with a single entry pointing to the new data record.

5. Forward, backward, and father pointers are set up as shown in the diagram below.

At this point, the creation of the new index level is complete. The BRA in the directory entry for the DAM file still points to the original index record, which is now the top of a two-level index.



The DIR entry points to the original index record (record 'I'), which now contains just pointers to records 'J' and 'K' -- the two records on the index level just created. Record 'J' contains the data record pointers originally in 'I' -- 'L', 'M', etc. Record 'K' contains a single pointer to the newly created data record 'N'.

Once an index level is created, it is never deleted until the file itself is deleted -- there will always be at least one record on each level. If the file is empty, there will be exactly one record on each index level. This is to avoid undue thrashing when records are being added and deleted near the threshold of an index's capacity. (The overhead of the "unnecessary" levels is only one record per level.)

APPENDIX F

OBSOLETE FILE SYSTEM SUBROUTINES

The subroutines described in this appendix are no longer in use by the current file management system. However, they are still in use by users who choose to continue using older versions of the file system and/or continue to use old style partitions. For this reason, they are collected and described here. These subroutines are:

ATTACH
CMREAD
CNAME$
COMINP
PRWFIL
RESTOR
RESUME
SAVE
SEARCH

****
* ATTACH *
****

The ATTACH subroutine has the same effect as the ATTACH internal command. The calling sequence is:

```
CALL ATTACH (Ufd, Ldisk, Password, Key, Altrtn)
```

To access files, the file system must be attached to some User File Directory (UFD). This implies that the file system has been supplied with the proper file directory name and either the owner or nonowner password, and the file system has found and saved the name and location of the file directory. After a successful attach, the name, location and owner/nonowner status of the UFD is referred to as the current UFD. As an option, this information may be copied to another place in the system, referred to as the home UFD. The ATTACH subroutine does not change the home UFD unless the user specifies to change it in the subroutine call. The user gets owner status if he gives the owner password, or gets nonowner status if he gives the nonowner password. The owner of a file directory can declare on a per-file basis what rights a nonowner has over the owner's files. The nonowner password may be given only under PRIMOS III and IV. (Refer to the description of the commands PASSWD and PROTECT of the PRIMOS Commands Guide (FDR3108) for more information.)

In attaching to a directory, the subroutine ATTACH specifies where to look for the directory. ATTACH specifies a User File Directory (UFD) in the Master File Directory (MFD) on a particular logical disk, a sub-directory in the current UFD, or the home UFD as the target-directory of the ATTACH operation. The format is:

```
CALL ATTACH (name, ldisk, password, key, altrtn)
```

KEY is composed of two subkeys that are combined additively: REFERENCE and SETHOME. All calls require a REFERENCE subkey. The REFERENCE subkeys are shown in the following table:

REFERENCE	Octal Value	Meaning
MFDUFD	Ø	Attach to NAME in MFD on LDISK
CURUFD	2	Attach to NAME in current UFD

The SETHOME subkeys are required on call; these subkeys are shown in the following table:

SETHOME	Octal Value	Meaning
---	Ø	Do not set home UFD to current UFD after attaching.
SETHOM	1	Set home UFD to current UFD after attaching.

The meaning of the remaining parameters on a call to ATTACH is as follows:

name — If the key is Ø and NAME is Ø, the home UFD is attached.

If the reference subkey is MFDUFD or CURUFD, NAME is either a six-character Hollerith expression or the name of a three-word array that specifies a Ufdname to be attached.

ldisk — If the reference subkey is MFDUFD, LDISK is the logical disk on which the MFD is to be searched for UFD NAME. LDISK must be a logical disk that has been started up by the STARTUP command. The special LDISK octal code 1ØØØØØ signifies: search all started-up logical devices in order Ø, 1, 2 ... n and attach to the UFD in which NAME appears in the MFD of the lowest numbered logical device. The special LDISK octal code 177777 signifies: search the MFD of the Ldisk currently attached to NAME.

If the reference subkey is CURUFD, or NAME is Ø, LDISK is ignored and is usually specified as Ø.

password — If the reference subkey is MFDUFD, CURUFD, or SEGUFD, PASSWORD is either a six-character Hollerith expression or the name of a three-word array that specifies one of the passwords of UFD NAME. If the password is blank, it is specified as three words of two blank characters.

altrtn An integer variable assigned the value of a label in the user's FORTRAN program, to be used as an alternate return in case of error. If this argument is Ø or omitted, an error message is printed and control returns to PRIMOS II or III.

A UFD attached through a segment directory reference does not have a name. On LISTF, such a UFD is listed with a name of six asterisks.

If an error is encountered and control goes to Altrtn, ERRVEC(1), a PRIMOS II vector, is set to the error type as follows:

Code	Message
AH	Name NOT FOUND
AL	No UFD ATTACHED
AR	Not a UFD (detected by PRIMOS III only)

A user obtains ERRVEC through a call to GETERR. The error 'Name NOT FOUND' is printed if one of the following errors occur:

1. key bad.
2. name is not found in the specified directory.
3. ldisk is out of range or not started up.
4. In a segment directory reference, NAME (1) is a closed unit or the unit is at end of file.

If the error BAD PASSWORD is obtained, the alternate return is never taken, and both the home UFD and current UFD are set to Ø to indicate that no UFD is attached. This feature is a system security measure to prevent a user from writing a program to try all possible passwords on a UFD.

Examples of ATTACH:

```
CALL ATTACH ('JHNDOE', -1, 'JJJ', Ø, ERR)
```

Searches for the UFD, JHNDOE, in the MFD (as specified in the Key) on the current logical device. If JHNDOE is found and the password, JJJ, matches the recorded password, then UFD JHNDOE is attached. The current UFD (now JHNDOE) is not set as the home UFD (as specified in the Key). The PRIMOS vector that points to the current UFD is set to this new directory.

****
* CMREAD *
****

Calling Sequence

```
CALL CMREAD  (ARRAY)
```

CMREAD reads 18 words (which represent the last command line input by the user) into the system vector ARRAY, as follows:

```
array(1)      Command  (or spaces)
array(2)
array(3)
array(4)
array(5)      name1  (or spaces)
array(6)
array(7)
array(8)      name2  (or spaces)
array(9)
array(10)     par1  (or zero)
array(11)     par2  (or zero)
    .
    .
    .
array(18)     par9  (or zero)
```

The command line may be accessed directly from array. name1 and name2 are normally UFD's or filenames, and par1 through par9 are octal numbers.

The last command line that has been input by the user is replaced by a new line of input by a call to the subroutines: COMANL, CNIN$ or T$AMLC. If none of these subroutines have been called before the CMREAD call, then CMREAD reads the last command line typed by the user or reads the last command line obtained through a command file.

****
* CNAME$ *
****

The CNAME$ routine allows the same action at user program level as the CNAME$ command allows at command level.

The calling sequence is:

```
CALL CNAME$ (oldnam, newnam, altrtn)
```

CNAME$ changes the name of Oldnam in the current UFD to Newnam. The user must have owner status to the UFD. The arguments are:

oldnam	A filename to be changed
newnam	The new filename for oldnam
altrtn	If not Ø, control goes to altrtn if any error occurs. If Ø, an error message is printed and control returns to PRIMOS.

If an error is encountered and control goes to altrtn, ERRVEC(1) is set to the error type as follows:

Code	Message
CA	newnam BAD NAME
CZ	newnam DUPLICATE NAME
SH	oldnam NOT FOUND
SI	oldnam IN USE
SL	NO UFD ATTACHED
SX	oldnam NO RIGHT

CNAME$ does not run under PRIMOS II.

****
* COMINP *
****

The COMINP routine allows the user to perform the same action at program level as the user command COMINPUT allows at command level. Refer to the PRIMOS Commands Guide (FDR3108) for details of the COMINPUT command. Briefly, COMINP causes PRIMOS to read input from a file rather than a terminal.

The calling sequence is:

```
CALL COMINP (name, funit, altrtn)
```

The arguments are:

name	Either a three-word array containing the filename of a command file, or the words TTY, CONTIN, or PAUSE.
funit	A File Unit (range 1 to 16; 1 to 15 under PRIMOS II) that is to be used for reading the command file.
altrtn	If not 0, control goes to altrtn in the event of an error while opening Name. If 0, an error message is printed and control returns to the operating system in the event of an error while opening Name.

If an error is encountered and control goes to Altrtn, ERRVEC(1) is set to the error type as follows:

Code	Message
SD	UNIT NOT OPEN
SH	name NOT FOUND
SI	name IN USE
SI	UNIT IN USE
SL	NO UFD ATTACHED
SX	name NO RIGHT

A user obtains ERRVEC through a call to GETERR.

****
* PRWFIL *
****

Definition of PRWFIL

PRWFIL is used to read, write, and position a file open on a file unit. A typical call to PRWFIL will read into a user buffer N words from a file open on Funit, starting at the file pointer in the file. A user may instead move the file pointer to an absolute position in the file. The two operations of reading-and-positioning or writing-and-positioning may be combined into a single call, with position occurring either before or after the read or write operation.

The calling sequence is:

```
CALL PRWFIL (key, funit, LOC (buffer), nwords, position, altrtn)
```

key is composed of three subkeys that are combined additively: rwkey, poskey, and mode. The poskey is required only on those calls in which positioning is requested. Subkeys with values of Ø may be omitted from the call. The PRWFIL call may be represented as:

```
CALL PRWFIL (rwkey+poskey+mode,funit,pbuffer,nwords,position,altrtn)
```

The rwkey subkeys are shown in the following table:

rwkey	Octal Value	Meaning
PREAD	1	Reads nwords from funit into buffer
PWRITE	2	Write nwords from buffer

The poskey subkeys are shown in the following table:

poskey	Octal Value	Meaning
PREREL	Ø	Moves the file pointer of funit position words relative to the current position before reading or writing
POSREL	2Ø	Moves the file pointer of funit position words relative to the current position after reading or writing
PREABS	1Ø	Moves the file pointer of funit to an absolute position specified by position(1) and position(2) before reading or writing
POSABS	3Ø	Moves the file pointer of funit to an absolute position specified by position(1)

and POSITION(2) after reading and writing

The MODE subkeys are shown in the following table:

MODE	Octal Value	Meaning
—	Ø	Reads or writes nwords
PCONV	4ØØ	Reads or writes a convenient number of words; less than or equal to nwords

The meaning of the remaining parameters in a call to PRWFIL are as follows:

funit — A file unit number 1 to 16 for PRIMOS III and IV (1 to 15 for PRIMOS II) upon which a file has been opened by a call to SEARCH or a command. PRWFIL actions are performed on this file unit.

buffer — Reading or writing is initiated at buffer. Note that buffer is obtained through the integer function LOC.

nwords — If the mode subkey is Ø, nwords is the number of words to be transferred to or from a file unit and a user buffer. If nwords is Ø, no words are transferred.

If the mode subkey is PCONV, NWORDS is the maximum number of words to be transferred. The number actually transferred is a number between 1 and nwords that is convenient and fast for PRWFIL to transfer. If NWORDS is Ø, no words are transferred. The user can establish how many words were transferred from ERRVEC(2).

For either mode, nwords may be between Ø and 65535.

position — If the POSKEY is PREREL or POSREL, POSITION is a single signed integer word for relative positioning. Positioning is forward and backward from the file pointer, depending on the POSITION sign. If position is Ø, no positioning is done.

If the key is PREABS or POSABS, position is a two-word integer array (V-record-number, word-number) for absolute positioning. If POSITION is (Ø,Ø) (both values Ø), the file pointer is moved to the beginning of the file.

altrtn An integer variable assigned the value of a label in the user's FORTRAN program to be used as an alternate return in case of uncorrectable errors. If the argument is Ø or omitted, an error message is printed and control returns to PRIMOS.

If an error is encountered and control goes to altrtn, ERRVEC(1) is set to the error type. This is a two-character code as follows:

Code	Message	Meaning
PD	PRWFIL UNIT NOT OPEN	Bad key, or file unit not open for read/write
PE	PRWFIL EOF	End-of-file reached on read or position
PG	PRWFIL BOF	Beginning of file reached on read or position
DJ	DISK FULL	No room left on disk

A user obtains ERRVEC through a call to GETERR, which is described in this section. A user may wish to handle one type of error and have the system type all other error messages and return to PRIMOS II or III. The user can tell PRERR to print the error message that would have been printed without altrtn.

On a PRWFIL EOF or PRWFIL BOF error, ERRVEC(2), is set to the number of words left to be transferred in the read or write requests. On all normal returns from PRWFIL, ERRVEC(3) and ERRVEC(4) are set to the file pointer of the file as a two-word array (record-number, word-number). On a call with the PCONV subkey, ERRVEC(2) is set to the number of words read.

On a DISK FULL error, the file pointer is set to the value it had at the beginning of the call. The user may, therefore, delete another file and restart the program by typing START. This feature works only with PRIMOS III and IV.

During the positioning operation PRWFIL, PRIMOS maintains a file pointer for every open file. Because a file may contain more than 65,535 words, the largest unsigned integer that can be represented in a 16-bit word, the file pointer occupies two words. The method of representation chosen is two words, the first of which is the V-record number and the second of which is a word number. Each V-record contains 44Ø words of data so the word number has a range of Ø to 439. The V-record number has a range of Ø to 32767. When a file is opened by a call to SEARCH, the file pointer is set so that the next word read is the first word of the file. The position pointer contains V-record Ø, word Ø, or briefly (Ø,Ø). If the user calls PRWFIL to read 49Ø words and does no positioning, at the end of the read operation the

file pointer is (V-record 1, word 50) or briefly (1,50). The V-record size (440) is constant for all disks and does not correspond to the physical record size.

A call to read or write N words causes N words to be transferred to or from the file, starting at the file pointer in the file. Following a call to transfer information, the file pointer is moved to the end of the data transferred in the file. Using POSKEY of PREABS or POSABS, the user may explicitly move the file pointer to (record number, word number) before or after the data transfer operation. Using a POSKEY of PREREL or POSREL, the user may explicitly move the file pointer forward position words from the current position, if position is positive. Using a POSKEY of PREREL or POSREL, the user may move the file point backward position words from the current position, if POSITION is negative. The maximum position that can be moved in the call is therefore plus or minus 32767 words. Positioning takes place before or after the data transfer, depending on the key. If nwords is 0 in any of the calls to PRWFIL, no data transfer takes place, so PRWFIL does only a pointer position operation. On normal returns from PRWFIL, ERRVEC (3) and ERRVEC (4) contain the file pointer as (record number, word number).

The mode subkey of PRWFIL is most frequently used to transfer a specific number of words on a call to PRWFIL. In these cases, the MODE is 0 and is normally omitted in PRWFIL calls. In some cases, such as in a program to copy a file from one file directory to another, a buffer of a certain size is set aside in memory to hold information, and the file is transferred a buffer full at a time. In the latter case, the user doesn't care how many words are transferred at each call to PRWFIL, so long as the number of words is less than the size of the buffer set aside in memory.

As the user would generally prefer to run his program as fast as possible, the PCONV subkey is used to transfer nwords, or less in the call to PRWFIL. The number of words transferred is a number convenient to the system, and therefore speeds up program run time. The number of words actually transferred is put in ERRVEC (2).

****
* RESTOR *
****

RESTOR has the same effect under program control as the RESTORE command.

The calling sequence is:

```
CALL RESTOR (vect, filename, altrtn)
```

RESTOR performs the inverse of the SAVE operation. The SAVEd parameters for a filename previously written to disk by SAVE are loaded into the nine-word array vect. The program itself is then loaded into high-speed memory, using the starting and ending address provided by VECT (1) and VECT (2).

If an error is encountered and control goes to altrtn, ERRVEC(1) is set to the error type as follows:

Code	Message
SH	Name NOT FOUND
SI	UNIT IN USE
SI	Name IN USE
SL	NO UFD ATTACHED
SX	NO RIGHT
PE	PRWFIL EOF

****
* RESUME *
****

RESUME has the same effect under program control as the RESUME command.

The calling sequence is:

```
CALL RESUME (filename)
```

****
* SAVE *
****

SAVE has the same effect under program control as the SAVE command.

The calling sequence is:

CALL SAVE (vect, filename)

The user sets up a nine-word vector vect before calling SAVE. vect(1) must be set to an integer which is the first location in memory to be saved, and vect(2) must be set to the last location to be saved. The rest of the vector may be set up at the programmer's option.

		Location
vect(3)	P Register	7
vect(4)	A Register	1
vect(5)	B Register	2
vect(6)	X Register	Ø
vect(7)	Keys	---
vect(8)	Spare	---
vect(9)	Spare	---

SAVE writes, to the named disk file, the nine-word vector vect, followed by the memory image starting at vect(1) and ending at vect(2).

****
* SEARCH *
****

Definition of SEARCH

SEARCH is used to connect a file to a file unit (open a file) or disconnect a file from a file unit (close a file). After a file is connected to a unit, PRWFIL and other routines may be called, either to position the current-position pointer of a file unit (file pointer) or to transfer information to or from the file (using the file unit to reference the file).

Opening a File

On opening a file, SEARCH specifies 1) allowable operations that may be performed by PRWFIL and other routines (these operations are read only, write only, or both read and write); 2) where to look for a file or where to add the file, if the file does not already exist; and 3) whether the file is to be opened for writing only or both reading and writing. SEARCH either specifies a filename in the currently attached user file directory or a file unit number on which a segment directory is open. In the segment directory reference, the file to be opened or closed has its beginning disk address given by the word at the current position pointer of the file unit.

SEARCH Actions

On creating a new file, the user specifies to SEARCH the file type of the new file.

The subroutine SEARCH may be used to perform actions other than opening and closing a file. SEARCH may delete a file, rewind a file unit, or truncate a file.

Upon opening a file, SEARCH sets the file pointer to the beginning of the file. Subroutines PRWFIL and others cause information to be transfered to or from the file unit, starting at the file pointer. After the transfer, the pointer is moved past the data transferred. A call to SEARCH to rewind a file causes the file pointer to be set to the beginning of the file. Subsequent calls to PRWFIL and other routines cause information transfer to occur as if the file had just been opened. A call to SEARCH to truncate a file causes all information beyond the file pointer to be removed from the file. This call is useful if one is overwriting a file with less information than was originally contained in the file.

Subroutine Call

SEARCH is used as in the following call:

Format:

```
CALL SEARCH (key, name, funit, altrtn)
```

key is composed of three subkeys that are combined additively: action, reference, and newfile. Not all subkeys are required on every call, and subkeys with values of zero can be omitted. The SEARCH call may therefore be represented as:

```
CALL SEARCH (action+reference+newfile, name, funit, altrtn)
```

All calls require an action subkey. The action subkeys are shown in the following table:

action	Octal Value	Meaning
OPNRED	1	Open name for reading on funit
OPNWRT	2	Open name for writing on funit
OPNBTH	3	Open name for both reading and writing on funit
CLOSE	4	Close file by name or by funit
DELETE	5	Delete file nam
EXIST	6	Check to see if file exists.
REWIND	7	Rewind file on funit
TRNCAT	10	Truncate file on funit
CNGACC	1000	Change access of file to funit

The reference subkeys are shown in the following table:

reference	Octal Value	Meaning
UFDREF	0	Searches for file name in the current user file directory (UFD) (as defined by a previous ATTACH) and perform the action in the action subkey on the specified file.
SEGREF	100	Performs the action specified in the action subkey on the file with the location indicated by the file pointer designated within the array name(1). This file unit must be an open segment directory.

Only those calls to SEARCH that reference a file in a UFD or segment directory need the reference key. Calls that reference file units do not need this key.

The following table lists the newfil subkeys:

newfil	Octal Value	Meaning
NTFILE	0	New threaded (SAM) file
NDFILE	2000	New directed (DAM) file
NTSEG	4000	New threaded (SAM) segment directory
NDSEG	6000	New directed (DAM) segment directory
NEWUFD	10000	New User File Directory (SAM)

Only those calls to SEARCH that generate a new file require a newfil subkey. On other calls, this subkey is ignored.

The name of the remaining parameters in a call to SEARCH are as follows:

name If the reference subkey is UFDREF, NAME is either a six-character Hollerith expression or the name of a three-word array that specifies a filename (existing or not).

If the reference subkey is UFDREF and name(1) is -1, the current UFD is opened. name = -1 must be used only in configuration with action subkeys 1, 2, or 3. Owner status of the current UFD is required.

If the reference subkey is SEGREF, name is a file unit(1-16; 1-15 under PRIMOS II) on which a segment directory is open.

On calls in which the action key requires only a file unit to specify the file to be acted on, name is ignored and, usually, specified as 0.

funit On calls that require a file unit number, funit is a number 1 to 16 (1-15 under PRIMOS II). On calls that require no unit number, funit is ignored and usually specified as 1.

altrtn altrtn is an integer variable assigned the value of a label return in the user's FORTRAN program to be used as an alternate in case of uncorrectable errors (e.g., attempting to open a file that is already open). If this argument is 0 or omitted, an error message is printed; control returns to PRIMOS if any error should occur while using SEARCH.

Error Messages

If an error is encountered and control goes to altrtn, ERRVEC(1) is set to a two-character code as follows:

Code	Message	Meaning
SA	BAD CALL TO SEARCH	Some parameter in call is invalid
SD	UNIT NOT OPEN	Attempt to truncate or rewind a file on a closed unit
SD	Name OPEN ON DELETE	Self-explanatory
SH	Name NOT FOUND	File Name not in UFD
SI	Name IN USE	File Name is already open
SI	UNIT IN USE	File unit is already open
SK	UFD FULL	Self-explanatory
SL	NO UFD ATTACHED	Self-explanatory
SQ	SEG-DIR ERROR	*SEG-DIR ERROR
SX	NO RIGHT	Access rights violation
DJ	DISK FULL	No room left on disk

*SEG-DIR ERROR:

Meaning

1. If attempting to open an existing file in the segment directory, *SEG-DIR ERROR means:

   a. The segment directory unit specified in NAME is not open for reading.

   b. The file pointer of the segment directory unit is at end of file, and therefore points to no disk address.

   c. The file pointer of the segment directory unit points to a Ø entry.

2. If attempting to open a new file in the current segment directory, *SEG-DIR ERROR means:

   The segment directory unit specified in NAME is not open for both reading and writing.

When a user obtains ERRVEC through a call to GETERR (described in this section), control is to go to altrtn. A user may wish to handle one type of error and have the system print all other error messages and return to PRIMOS. The user can call PRERR to print the error message that would have been printed without altrtn.

ERRVEC(2) is set to a file type on a normal return of a call to SEARCH to open a file, using action keys of OPNRED, OPNWRT, or OPNBTH. The codes are:

ERRVEC(2)	File Type
Ø	Threaded file (SAM)
1	Directed file (DAM)
2	Threaded segment directory (SAM)
3	Directed segment directory (DAM)
4	User File Directory (SAM)

Access Rights and Call to SEARCH

Under PRIMOS III and IV, the access rights of files are checked when a user attempts to open a file through a call to SEARCH. Under PRIMOS II, access rights are not checked.

A SEARCH call that creates a new file gives that file default access rights. Defaults access rights are: owner has all rights; nonowner has no rights.

Adding and Deleting Files

For references to user file directories, a call to SEARCH to open a file for writing or both reading and writing causes SEARCH to look in the current User File Directory for the file. If the file is not found in the UFD, the file name and beginning disk address of a new file is appended to the UFD, and the new file is opened for the appropriate acitvity. A call to delete a file from a UFD removes the name and beginning disk address from the UFD and shortens the UFD.

For references to segment directories, a call to SEARCH to open a file for writing or reading and writing causes SEARCH to examine the word at the file pointer of the referenced segment directory file unit. If the word is not zero, SEARCH considers the word to be a beginning record address of an already created file. SEARCH opens the file for writing or reading and writing. If the word is zero, SEARCH writes the beginning disk address of a new file in that word and opens the file. If the file pointer is positioned at the end of file, the file is lengthened one word and SEARCH writes the beginning disk address of a new file in that word, and opens the file. A call to delete a file from a segment directory causes the beginning disk address of a file at the file pointer of the segment directory to be replaced by zero. The segment directory is not shortened. An attempt to open a file for reading in a segment directory when its file pointer points to zero or is at end-of-file generates a SEG-DIR error. In no case is the file pointer of a segment directory moved.

Closing and Opening Files

On a call to close a file, SEARCH attempts to close file NAME and generates an error message or goes to the alternate return if NAME is not found. FUNIT is ignored unless NAME is Ø. If NAME is Ø, SEARCH ensures that FUNIT is closed. That is, it closes FUNIT if FUNIT is open but does not generate an error message if the file unit is closed.

Example:

```
CALL SEARCH  (1, 'OBJECT', 1, $50)
```

Searches for a file, OBJECT, in the current UFD and opens it for reading; if file is not found, return via statement 50 is made.

The user is allowed to open the current UFD for reading via a call to SEARCH. The calling sequence for this feature is:

```
CALL SEARCH  (1, -1, Funit, Altrtn)
```

This call opens the current UFD for reading on Funit. The user must have owner access rights to the UFD; i.e., the owner password must have been given in the most recent call to ATTACH (or ATTACH command). Control goes to Altrtn if there is no UFD attached, if Funit is already in use, or if the user does not have owner rights to the UFD.

Changing the Access of a File

A user may change the access of a file that is open on FUNIT to OPNREAD, OPNWRT, or OPNBTH.

Example:

```
CALL SEARCH (CNGACC + OPNWRT, 0, FUNIT, 0)
```

Access rights are checked to determine if the user has a right to accomplish the requested operations.

Checking the Existence of a File

If the user desires to find out if a certain file exists in the current UFD, the user can call SEARCH with the EXIST key. The file unit should be specified as 1. The file is not affected in any way and access rights are not checked.

Sharing Files

Two or more users may be attached to the same UFD at the same time. Furthermore, two or more users may have the same file open for reading, and they may be reading from the same file at the same time. File interlocks are provided to prevent one user from opening the file for reading or writing while another user has the file open for writing. File interlocks also prevent one user from opening the file for writing while another user has the file open for reading. If these interlock situations are detected by SEARCH, the user gets the error message: FILE IN USE. The file interlocks also apply to the case of the same user attempting to open the file on different file units (FUNITS).

APPENDIX G

ERROR MESSAGES AND CODES (SYSCOM>ERRD.F)

INTRODUCTION

This appendix defines PRIMOS error messages and codes.

```
C ERRD.F, SYSCOM, OS GROUP, Ø3/29/79
C    MNEMONIC CODES FOR FILE SYSTEM (FTN)
C    Copyright 1978, Prime Computer, Inc., Wellesley, MA
      NOLIST
C
C     TABSET 6 11 23 56 65
C
      INTEGER*2 E$EOF,E$BOF,E$UNOP,E$UIUS,E$FIUS,E$BPAR,E$NATT,
     X        E$FDFL,E$DKFL,E$NRIT,E$FDEL,E$NTUD,E$NTSD,E$DIRE,
     X        E$FNTF,E$FNTS,E$BNAM,E$EXST,E$DNTE,E$SHUT,E$DISK,
     X        E$BDAM,E$PTRM,E$BPAS,E$BCOD,E$BTRN,E$OLDP,E$BKEY,
     X        E$BUNT,E$BSUN,E$SUNO,E$NMLG,E$SDER,E$BUFD,E$BFTS,
     X        E$FITB,E$NULL,E$IREM,E$DVIU,E$RLDN,E$FUIU,E$DNS,
     X        E$TMUL,E$FBST,E$BSGN,E$FIFC,E$TMRU,E$NASS,E$BFSV,
     X        E$SEMO,E$NTIM,E$FABT,E$FONC,E$NPHA,E$ROOM,E$ITRE,
     X        E$WTPR,E$FAMU,E$TMUS,E$NCOM,E$NFLT,E$STKF,E$STKS,
     X        E$NOON,E$CRWL,E$CROV,E$CRUN,E$CMND,E$RCHR,E$NEXP,
     X        E$BARG,E$CSOV,E$NOSG,E$TRCL,E$NDMC,E$DNAV,E$DATT,
     X        E$BDAT,E$BLEN,E$BDEV,E$QLEX,E$NBUF,E$INWT,E$NINP,
     X        E$DFD, E$DNC, E$SICM,E$SBCF,E$VKBL,E$VIA, E$VICA,
     X        E$VIF, E$VFR, E$VFP, E$VPFC,E$VNFC,E$VPEF,E$VIRC,
     X        E$IVCM, E$DNCT, E$BNWD,
     X        E$LAST
C
      PARAMETER
     X
     X /*************************************************************/
     X /*                                                           */
     X /*                                                           */
     X /*      CODE DEFINITIONS                                     */
     X /*                                                           */
     X /*                                                           */
     X    E$EOF = 1,  /* END OF FILE                    PE        */
     X    E$BOF = 2,  /* BEGINNING OF FILE              PG        */
     X    E$UNOP= 3,  /* UNIT NOT OPEN                  PD,SD     */
     X    E$UIUS= 4,  /* UNIT IN USE                    SI        */
     X    E$FIUS= 5,  /* FILE IN USE                    SI        */
     X    E$BPAR= 6,  /* BAD PARAMETER                  SA        */
     X    E$NATT= 7,  /* NO UFD ATTACHED                SL,AL     */
     X    E$FDFL= 8,  /* UFD FULL                       SK        */
     X    E$DKFL= 9,  /* DISK FULL                      DJ        */
```

```
X    E$NRIT=1Ø,  /* NO RIGHT                          SX       */
X    E$FDEL=11,  /* FILE OPEN ON DELETE               SD       */
X    E$NTUD=12,  /* NOT A UFD                         AR       */
X    E$NTSD=13,  /* NOT A SEGDIR                      —        */
X    E$DIRE=14,  /* IS A DIRECTORY                    —        */
X    E$FNTF=15,  /* (FILE) NOT FOUND                  SH,AH    */
X    E$FNTS=16,  /* (FILE) NOT FOUND IN SEGDIR        SQ       */
X    E$BNAM=17,  /* ILLEGAL NAME                      CA       */
X    E$EXST=18,  /* ALREADY EXISTS                    CZ       */
X    E$DNTE=19,  /* DIRECTORY NOT EMPTY               —        */
X    E$SHUT=2Ø,  /* BAD SHUTDN (FAM ONLY)             BS       */
X    E$DISK=21,  /* DISK I/O ERROR                    WB       */
X    E$BDAM=22,  /* BAD DAM FILE (FAM ONLY)           SS       */
X    E$PTRM=23,  /* PTR MISMATCH (FAM ONLY)           PC,DC,AC */
X    E$BPAS=24,  /* BAD PASSWORD (FAM ONLY)           AN       */
X    E$BCOD=25,  /* BAD CODE IN ERRVEC                —        */
X    E$BTRN=26,  /* BAD TRUNCATE OF SEGDIR            —        */
X    E$OLDP=27,  /* OLD PARTITION                     —        */
X    E$BKEY=28,  /* BAD KEY                           —        */
X    E$BUNT=29,  /* BAD UNIT NUMBER                   —        */
X    E$BSUN=3Ø,  /* BAD SEGDIR UNIT                   SA       */
X    E$SUNO=31,  /* SEGDIR UNIT NOT OPEN              —        */
X    E$NMLG=32,  /* NAME TOO LONG                     —        */
X    E$SDER=33,  /* SEGDIR ERROR                      SQ       */
X    E$BUFD=34,  /* BAD UFD                           —        */
X    E$BFTS=35,  /* BUFFER TOO SMALL                  —        */
X    E$FITB=36,  /* FILE TOO BIG                      —        */
X    E$NULL=37,  /* (NULL MESSAGE)                    —        */
X    E$IREM=38,  /* ILL REMOTE REF                    —        */
X    E$DVIU=39,  /* DEVICE IN USE                     —        */
X    E$RLDN=4Ø,  /* REMOTE LINE DOWN                  —        */
X    E$FUIU=41,  /* ALL REMOTE UNITS IN USE           —        */
X    E$DNS =42,  /* DEVICE NOT STARTED                —        */
X    E$TMUL=43,  /* TOO MANY UFD LEVELS               —        */
X    E$FBST=44,  /* FAM - BAD STARTUP                 —        */
X    E$BSGN=45,  /* BAD SEGMENT NUMBER                —        */
X    E$FIFC=46,  /* INVALID FAM FUNCTION CODE         —        */
X    E$TMRU=47,  /* MAX REMOTE USERS EXCEEDED         —        */
X    E$NASS=48,  /* DEVICE NOT ASSIGNED               —        */
X    E$BFSV=49,  /* BAD FAM SVC                       —        */
X    E$SEMO=5Ø,  /* SEM OVERFLOW                      —        */
X    E$NTIM=51,  /* NO TIMER                          —        */
X    E$FABT=52,  /* FAM ABORT                         —        */
X    E$FONC=53,  /* FAM OP NOT COMPLETE               —        */
X    E$NPHA=54,  /* NO PHANTOMS AVAILABLE             -        */
X    E$ROOM=55,  /* NO ROOM                           —        */
X    E$WTPR=56,  /* DISK WRITE-PROTECTED              JF       */
X    E$ITRE=57,  /* ILLEGAL TREENAME                  FE       */
X    E$FAMU=58,  /* FAM IN USE                        —        */
X    E$TMUS=59,  /* MAX USERS EXCEEDED                —        */
X    E$NCOM=6Ø,  /* NULL_COMLINE                      —        */
X    E$NFLT=61,  /* NO_FAULT_FR                       —        */
X    E$STKF=62,  /* BAD STACK FORMAT                  —        */
```

```
X     E$STKS=63,  /* BAD STACK ON SIGNAL              —        */
X     E$NOON=64,  /* NO ON UNIT FOR CONDITION         —        */
X     E$CRWL=65,  /* BAD CRAWLOUT                     —        */
X     E$CROV=66,  /* STACK OVFLO DURING CRAWLOUT      —        */
X     E$CRUN=67,  /* CRAWLOUT UNWIND FAIL             —        */
X     E$CMND=68,  /* BAD COMMAND FORMAT               —        */
X     E$RCHR=69,  /* RESERVED CHARACTER               —        */
X     E$NEXP=70,  /* CANNOT EXIT TO COMMAND PROC      —        */
X     E$BARG=71,  /* BAD COMMAND ARG                  —        */
X     E$CSOV=72,  /* CONC STACK OVERFLOW              —        */
X     E$NOSG=73,  /* SEGMENT DOES NOT EXIST           —        */
X     E$TRCL=74,  /* TRUNCATED COMMAND LINE           —        */
X     E$NDMC=75,  /* NO SMLC DMC CHANNELS             —        */
X     E$DNAV=76,  /* DEVICE NOT AVAILABLE             DPTX     */
X     E$DATT=77,  /* DEVICE NOT ATTACHED              —        */
X     E$BDAT=78,  /* BAD DATA                         —        */
X     E$BLEN=79,  /* BAD LENGTH                       —        */
X     E$BDEV=80,  /* BAD DEVICE NUMBER                —        */
X     E$QLEX=81,  /* QUEUE LENGTH EXCEEDED            —        */
X     E$NBUF=82,  /* NO BUFFER SPACE                  —        */
X     E$INWT=83,  /* INPUT WAITING                    —        */
X     E$NINP=84,  /* NO INPUT AVAILABLE               —        */
X     E$DFD =85,  /* DEVICE FORCIBLY DETACHED         —        */
X     E$DNC =86,  /* DPTX NOT CONFIGURED              —        */
X     E$SICM=87,  /* ILLEGAL 3270 COMMAND             —        */
X     E$SBCF=88,  /* BAD 'FROM' DEVICE                —        */
X     E$VKBL=89,  /* KBD LOCKED                       —        */
X     E$VIA =90,  /* INVALID AID BYTE                 —        */
X     E$VICA=91,  /* INVALID CURSOR ADDRESS           —        */
X     E$VIF =92,  /* INVALID FIELD                    —        */
X     E$VFR =93,  /* FIELD REQUIRED                   —        */
X     E$VFP =94,  /* FIELD PROHIBITED                 —        */
X     E$VPFC=95,  /* PROTECTED FIELD CHECK            —        */
X     E$VNFC=96,  /* NUMERIC FIELD CHECK              —        */
X     E$VPEF=97,  /* PAST END OF FIELD                —        */
X     E$VIRC=98,  /* INVALID READ MOD CHAR            —        */
X     E$IVCM=99,  /* INVALID COMMAND                  —        */
X     E$DNCT=100, /* DEVICE NOT CONNECTED             —        */
X     E$BNWD=101, /* BAD NO. OF WORDS                 —        */
X     E$LAST=101  /* THIS ***MUST*** BE LAST          —        */
X /*                                                           */
X /*                                                           */
X /*************************************************************/
 LIST
```

NEW FILE SYSTEM ERROR HANDLING CONVENTIONS

Motivation

All the file management system routines described in Section 3 employ error handling procedures that are standard to PRIMOS subsystems. The

error handling facilities do not affect previously existing programs. Only programs using the file management system calls need to be aware of the error handling described in this section.

The error handling protocol was motivated by the following considerations.

1.\Except for a few restricted cases, FORTRAN non-local GOTOs do not work in 64V mode.

2.\Non-local GOTOs are a violation of good programming practice.

3.\Error information in a recursive/reentrant environment must be associated with a particular call, not left in a single static place (e.g., ERRVEC).

The Return Code Parameter

All error codes, formerly placed in ERRVEC, are now returned to the user in a 16-bit user-supplied integer variable. For example, in the call:

```
CALL PRWF$$ (KEY,UNIT,LOC(BFR),NW,POS,RNW,CODE)
```

CODE is an integer that PRWF$$ sets to the appropriate return code.

CODE can be thought of as a replacement for the (optional) alternate-return argument.

The effect of the old error handling scheme can be achieved through code such as:

```
CALL CREA$$ (NAME,NAMLEN,OPASS,NPASS,CODE)
IF (CODE.NE.Ø) GOTO 99
```

which would be equivalent to supplying an alternate return (ALTRTN) of $99 with old partitions (except, of course, that the subroutine GETERR need not be called to obtain the error code). Note that CODE should always be checked for zero or non-zero to ensure that errors do not go unnoticed.

STANDARD SYSTEM ERROR CODE DEFINITIONS

Standard system error codes are FORTRAN PARAMETER or PMA EQU variables with standardized names. In all cases, zero means no error. Any other value identifies a particular error or exceptional (not necessarily error) condition. All reference to specific code values (other than zero) should be by the standardized names. For convenience, all names are defined in two $INSERT files, ERRD.F for FORTRAN and ERRD.P for PMA. These files are included in the UFD SYSCOM on Volume 1 of the master disk.

ERROR HANDLING ROUTINE

The following routine, ERRPR$, provides all the new error handling facilities.

ERRPR$ -- Print Standard System Error Message

ERRPR$ interprets a return code and, if non-zero, prints a standard message followed by optional user text.

Calling Sequence

CALL ERRPR$ (key,code,text,txtlen,name,namlen)

Parameters

key — An integer specifying the action to take subsequent to printing the message. Possible values are:

- K$NRTN — Exit to the system, never return to the calling program.
- K$SRTN — Exit to the system, return to the calling program following an 'S' command.
- K$IRTN — Return immediately to the calling program.

code — An integer variable containing the return code from the routine that generated the error.

text — A message to be printed following the standard error message. Text is omitted by specifying both text and txtlen as Ø.

txtlen — The length in characters of text.

name — The name of the program or subsystem detecting or reporting the error. name is omitted by specifying both name and namlen as Ø.

namlen — The length in characters of name.

Notes on Usage

If code is Ø, no printing occurs, and ERRPR$ immediately returns to the calling program. The format of the message for non-zero values of CODE is:

<standard text>. <user's text if any> (<name if any>)

The system standard text associated with code is not preceded by any newline characters or blanks and ends with a period. If txtlen is greater than zero, this is followed by a blank followed by no more than 64 characters of text. If namlen is greater than zero, this is followed by a blank and no more than 64 characters of name enclosed in parentheses. The line is terminated with a newline.

If ERRPR$ is called with the special error code E$NULL, no system message is printed. Other parameters behave normally.

If ERRPR$ is called with an unrecognized value of code, the standard system message is 'ERROR=ddddd', where ddddd is the decimal value of code. This can be used to display user-defined errors. User defined errors should use codes above 1ØØØØ.

Examples

Following a call to PRWF$$, if CODE=E$UNOP, the call

```
CALL ERRPR$ (K$SRTN,CODE,'DO A STATUS',11,'PRWF$$',6)
```

would result in the message:

UNIT NOT OPEN. DO A STATUS (PRWF$$)

To print a user-defined error message:

```
CALL ERRPR$ (K$IRTN,1Ø328,'MY MESSAGE',1Ø,Ø,Ø)
```

will print:

ERROR=1Ø328. MY MESSAGE

SUBROUTINE NAME INDEX