PRINTING HISTORY — Subroutines Reference Guide

Edition	Date	Number	Software Release
*First Edition	March 1979	PDR3621	16.3
Second Edition	January 1980	PDR3621	17.2
Update 1	December 1980	PTU2600-078	18.1
Third Edition	July 1982	DOC3621-190	19.0
Update 1	April 1985	UPD3621-31A	19.4

*This edition is out of print.

A vertical bar in the margin of the Table of Contents indicates an addition since the last printing.

CUSTOMER SUPPORT CENTER

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)	1-800-541-8888 (within Alaska)
1-800-343-2320 (within other states)	1-800-651-1313 (within Hawaii)

HOW TO ORDER TECHNICAL DOCUMENTS

Obtain an order form, a catalog, and a price list from one of the following:

Inside U.S.

Software Distribution
Prime Computer, Inc.
74 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053

Outside U.S.

Contact your local Prime
subsidiary or distributor.

Contents

PART V - INPUT/OUTPUT LIBRARY SUBROUTINES

PART VI - COMMUNICATIONS CONTROLLERS AND REALTIME SUBROUTINES

PART VII - CONDITION HANDLING

PART VIII - LIBRARY BUILDING AND MANAGEMENT

APPENDIXES

INDEXES

About This Book

This book describes the subroutines that can be called from Prime's high-level languages or the Prime Macro Assembler (PMA). It also discusses how to call these subroutines from languages supported by Prime.

Procedures relating to building and modifying libraries and changing Input/Output Control System device assignments are included for user convenience. Use of Prime's condition mechanism is discussed in detail. An overview of pre-Rev. 19 PRIMOS file system concepts and usage is in Appendix I.

SUGGESTED REFERENCES

The Prime User's Guide (PDR4130) contains information on system use, directory structure, the condition mechanism, CPL files, ACLs, global variables, and how to load and execute files with external subroutines. Language programmers will also need the reference guide for their particular language. Programmers who wish more advanced information on library management or I/O manipulation should consult the System Administrator's Guide (PDR3109).

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id
underlining in examples	In examples, user input is underlined but system prompts and output are not.	OK, SEG -LOAD
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items (0-n).	CALL xxx (key [,altrtn])
Braces { }	Braces enclose a vertical list of items. Choose one and only one of these items.	CALL $\left\{\begin{matrix}\text{CLINEQ}\\ \text{LINEQ}\\ \text{DLINEQ}\end{matrix}\right\}$
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	CALL TIMDAT(array, n)
Hyphen -	Wherever a hyphen appears in a command line option, it is a required part of that option.	SPOOL -LIST

ADDITIONAL DOCUMENTATION CONVENTIONS

Notation Conventions

Convention	Explanation	Example
Angle Brackets < >	Angle brackets must be used as shown to separate the elements of a pathname.	<FOREST>BEECH>LEAF4
Colon :	A colon before a number indicates that octal notation follows.	:100
Apostrophe '	An apostrophe before a number indicates that octal notation follows.	'100

Filename Conventions

Convention	Explanation
filename.languagename or filename	Source file (for example, MYPROG.FTN)
filename.BIN or B_filename	Binary (object) file
filename.LIST or L_filename	Listing file
filename.SEG or #filename	Saved executable runfile (V-mode)
filename.SAVE or *filename	Saved executable object image (R-mode)

Filenames may be comprised of 1 to 32 characters inclusive, the first character of which must be nonnumeric. Names should not begin with a hyphen (-) or underscore (_). Filenames may be composed only of the following characters: A-Z, 0-9, _ # $ & - * . and /.

See the manual for each language for an explanation of how the various names for source, object, listing, and runtime files relate to each other. A general explanation is also in the *Prime User's Guide*.

Note

On some devices, the underscore (_) may print as back arrow (←).

PART I
Overview

1 Introduction

DOCUMENT ORGANIZATION

This guide is divided into eight parts which are detailed in the Table of Contents. They cover the following topics:

I	Overview
II	Language interfaces to standard subroutines
III	PRIMOS subroutines
IV	Math, applications, and sort library subroutines
V	Input/output library subroutines
VI	Subroutines that support communications controllers and semaphores
VII	Subroutines that support the condition-handling mechanism
VIII	Library management for object libraries

In addition, the Appendixes contain tables, new Rev. 19 subroutines, and some information of use only for revisions of PRIMOS before 19. There is a general index, and also an index of subroutine names only.

MAJOR CHANGES IN THE REV. 19 SUBROUTINE DOCUMENTATION

Chapters 1 and 2 have been rewritten. Chapters 3 through 8, the language interfaces, have been added. These additions have caused old Chapters 3 through 17 to be renumbered and, in some cases, reorganized. Old Chapter 3 is incorporated into Appendix I. Chapters 21 (SEMAPHORES AND TIMERS) and 23 (CONDITION-MECHANISM SUBROUTINES) have been rewritten. Appendixes A, B, and K have been added. The index of subroutines by name has been expanded to include a one-line description of each subroutine. In chapters not mentioned above as new or rewritten, change bars in the margin mark significant changes in content.

The chapters and appendixes have been renumbered as follows:

Old	New
1	1
2	2
3	Appendix I
4	9
5	10
6	5
7	FORTRAN guides
8	FORTRAN guides
9	Appendix G
10	11
11	12
12	13
13	14, 15
14	17, Appendix E
15	16
16	14, 17, 18, 19
17	17
18	18
19	19
20	20
21	21
22	23
23	22
A	F
B	J
C	H
D	C
E	I
F	Deleted
G	D

The following subroutine descriptions have been added in this edition of the Subroutines Reference Guide:

- The new ACLs, file maintenance, and date-retrieval subroutines in Appendix A.
- The message-support subroutines in Appendix B.
- APSFX$ - Append a suffix to a pathname.
- ASNLN$ - Assign AMLC line.
- CL$PIX - Parse command line.
- FNCHK$ - Check a filename for valid format.
- GCHAR - Get a character from an array.
- GV$GET - Retrieve the value of a global variable.
- GV$SET - Set the value of a global variable.
- I$AA12 - Read ASCII from terminal or input stream.
- IDCHK$ - Check an id for valid format.
- LON$CN - Enable or disable logout notification.
- LON$R - Retrieve logout notification information.
- MKON$P - Create an on-unit from F77 or PL1G.
- MRG2$S - Return next merged record.
- MRG3$S - Close merged input files.
- PHNTM$ - Start a phantom.
- PWCHK$ - Check a password for valid format.
- Q$READ - Read quota information.
- Q$SET - Set quota maximum.
- SCHAR - Store a character in an array.
- SEM$CL - Close named semaphore.
- SEM$OU - Open named semaphore by file unit.

- SEM$TW - Timed wait for named semaphore.
- SRSFX$ - Search for a file with any of a list of suffixes.
- TNCHK$ - Check a pathname for valid format.

WHAT IS NOT IN THIS BOOK

Only subroutines that are useful for programmers are discussed in this guide. Libraries such as COBOL (VCOBLB), RPG (RPGLIB), or PL1G (PL1GLB) contain subroutines that are used exclusively by the appropriate compiler. The use of these libraries is not discussed here, nor is the use of FORTRAN library subroutines such as IFIX or INT that are generated and used only by the FORTRAN compiler. Thus, old Chapters 7 and 8 have been omitted, and the material is in the relevant FORTRAN guide. In addition, the obsolete subroutines ATTACH, CMREAD, CNAME$, COMINP, PRWFIL, RESTOR, RESUME, SAVE, and SEARCH have been deleted.

2
Overview of Subroutines

OVERVIEW OF SUBROUTINE USE

This is a reference guide and is intended for users who already know how to call subroutines from a high-level language or from PMA. The following overview merely summarizes conventions for calling subroutines. For more information, see the chapter on your particular language.

A subroutine is a module of code that may be called from another module. It is useful for performing operations that cannot be performed by the calling language, or for performing standard operations faster. Users may write their own subroutines to supply customized or repetitive operations. However, this guide discusses only standard subroutines provided with the PRIMOS operating system or in standard libraries.

Functions and Subroutines

In this guide, a function is a call that returns a value. It must be called by being assigned to a variable, for example:

```
    VALUE1 = DELE$A(arg1, arg2)
```

A subroutine returns values only through its arguments. It is called this way:

```
CALL GV$GET(arg1, arg2, arg3, arg4)
```

However, the word subroutine is also used as the collective term for both of these modules.

Direct Entry Calls

All recent standard subroutines are direct entry calls. A direct entry involves execution of a routine within PRIMOS, the Prime operating system. The library call in this case contains only an interlude or call to the PRIMOS routine. This feature is of direct use only to the PMA programmer, who may use the PCL (procedure call) instruction rather than CALL to call the subroutines. For programmers in all languages, the feature means that repeated calls to the subroutine are faster, but the call is only available in V-mode and I-mode. A list of direct calls is supplied in Chapter 8.

Subroutine Arguments

Subroutines usually expect one or more arguments from the calling program. These arguments must be of the data type expected, and be passed in the order expected. Table 3-1 in Chapter 3 shows how a data type named in one language should be described in your calling language in order to be acceptable to the subroutine. All standard Prime subroutines are written in FORTRAN, PMA, or a system version of PL/I Subset G (PL1G). Chapters 3 through 8 discuss how to translate the data types expected by these languages into other Prime languages.

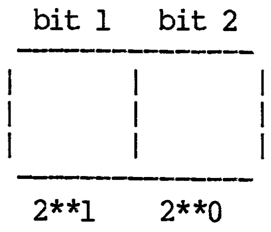
It is necessary, however, that arguments be passed in the same order as expected by execution. If too few arguments are passed, execution causes an error message such as POINTER FAULT or ILLEGAL SEGNO. If too many arguments are passed, the subroutine ignores the extra arguments, but will probably perform incorrectly.

How to Set Bits in Arguments

Sometimes a subroutine expects an argument that consists of a number of bits that must be set on or off.

A data item is stored in a computer as a collection of bits, which can each have one of two values, off or on. On Prime computers, off is arbitrarily equated to 0 or false, and on is equated to 1 or true. (This is not the same as the FORTRAN values .FALSE. and .TRUE., which are the logical data type.) When bits are stored as part of a group, the position of the bit gives it another value in addition to 1 or 0.

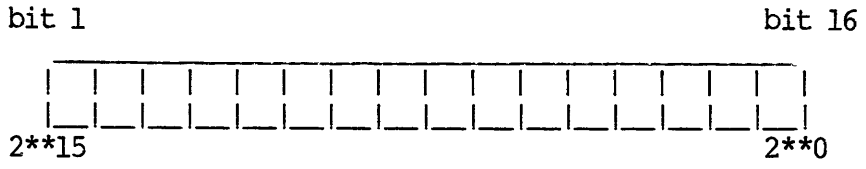
Its position equates it to a power of 2. Consider an argument that contained only two bits, represented in Figure 2-1.



Values of Bit Positions -- Two Bits
Figure 2-1

The low-order bit would be in the position of 2 to the 0 power, and its value, if ON, would be 1. The high-order bit would be in the position of 2 to the first power, and its value, if ON, would be 2. (If OFF, the value of a bit is always 0.) By convention, the low-order bit is called the rightmost bit and the high-order bit is called the leftmost bit.

In an argument containing 16 bits, choose the bits that you want to set ON, compute their value by position, and add these values. The resulting decimal value is what you should assign to the subroutine argument for the options you want. For example, if you want to set the sixteenth and the seventh bit, compute 2 to the 0 power plus 2 to the ninth power, which amounts to 1 plus 512, or 513. Figure 2-2 illustrates values of bit positions in a 16-bit argument.



Values of Bits in a 16-bit Argument
Figure 2-2

Key Names and Code Names

Many subroutine descriptions in this guide use data names for numeric values. These names are in the form x$yyyy, where x is either K, A, or E, and yyyy is a combination of letters. Examples are:

```
K$CURR
A$DEC
E$FNTF
```

The values of these keys are included in various files in the UFD called SYSCOM. It is recommended that programs use these data names rather than the numeric values for clarity. How to insert the key values in a program is discussed for each language in Chapters 3 through 8.

Loading Subroutines

A subroutine may be written in a different language from that of the calling program; in any case, the call only causes the object or binary code to be called. This code is in machine language, as is the object code that calls it at runtime. In PRIMOS, all subroutines must be loaded in the runtime module (memory image) in order to be found when they are called. Loading is done with the SEG utility for V-mode, and with the LOAD utility for R-mode. All object files loaded into one runtime file must be in the same mode, which means that not all subroutines can be used with all languages. Loading of all system subroutines in the FTNLIB, PFTNLIB, and PRIMOS libraries is done with the LI command of SEG or LOAD, with no operands. Loading of subroutines in the other libraries must be done by the programmer with the command LI plus the library name after SEG or LOAD is invoked. Examples of the loading process are given in Chapters 3 through 8.

If you try to execute a program that calls subroutines and get a runtime error message, reload and, after the LI command, use the MAP 3 command to see whether any missing subroutine names are displayed. If necessary, refer to the section on LOCATION OF LIBRARIES below to find where the missing subroutine is stored. (MAP 3, along with other load options, is explained in detail in the LOAD and SEG Reference Guide).

The loading process is different for BASIC/VM, which takes care of editing, compiling, loading, and execution within the special environment it creates.

The examples at the end of Chapters 3 through 8 show how to load programs that include subroutines.

LOCATION OF LIBRARIES

The object code for the standard library subroutines is contained in the UFD named LIB, and is loaded with the command LI for LOAD or SEG. Other libraries such as VSRTLI and VAPPLB must be loaded separately with the LI command followed by the library name. To get a list of all the libraries in the UFD LIB, use the commands:

```
ATTACH LIB
LISTF  (or LD for an alphabetical listing)
```

The libraries described in this guide are:

Library	R-mode File	V-mode File
PRIMOS including file system, condition mechanism, controllers, semaphore handlers, and IOCS	LIB>FTNLIB.BIN	LIB>PFTNLB.BIN
Application	LIB>APPLIB.BIN	LIB>VAPPLB.BIN
In-memory sorts	LIB>MSORTS>BIN	LIB>VMSORT.BIN
Matrix	LIB>MATHLB.BIN	not available
Sort	LIB>SRTLIB.BIN	LIB>VSRTLI.BIN
Spool	LIB>SPOOL$.BIN	LIB>VSPOO$.BIN

Note

The R-mode libraries are not being updated. Newer subroutines (such as GPATH$ or LOGO$$) are in the V-mode libraries only.

FTNLIB.BIN has been duplicated as SVCLIB.BIN.

There are other libraries in LIB that are not described in this guide. The subroutines in some of these libraries, such as PRIMENET, FORTRAN, the Block Device Interface, (BDVLIB), and MIDAS (KIDALB and VKDALB) are discussed in other guides. The calls to subroutines in other libraries, such as RPG, are generated automatically by compilers. The details need never concern the programmer.

OVERVIEW OF THE LIBRARIES

FORTRAN Library

The FORTRAN library contains many subroutines that are discussed in the following sections, such as the IOCS library. However, this library is also very important because it is the basis for most other libraries, including language libraries. This is why, except with PMA, loading of any program usually includes the command LI with no operand, whether the program contains subroutine calls or not. The command LI loads the FORTRAN library and checks that all subroutines called are present.

The FORTRAN library file also contains FORTRAN function subroutines and math subroutines. They are described in the FORTRAN Reference Guide and the FORTRAN 77 Reference Guide.

The FORTRAN library also contains arithmetic subroutines that the FORTRAN compiler uses. Some of these subroutines can also be called from PMA. These routines perform arithmetic operations on single-precision integers, single- and double-precision floating point, and complex numbers. They are listed in Appendix F.

File-handling Subroutines

All file handling is done by a collection of special subroutines, some internal to PRIMOS, and others available as application library routines. PRIMOS file-handling subroutines are described in Chapter 9.

All the PRIMOS file-handling subroutines called by the user are loaded with the FORTRAN library.

File Handling in User Programs: The file-handling subroutines simplify communication between the PRIMOS file structure and user programs. They can be used, for example, to verify the existence of a file before the program accesses it, to delete a file, or to check for a valid filename entered by a user.

Many of these subroutines allow a program to access files directly through file unit numbers, which is faster than access by filenames. File units are explained in Chapter 9. For example, at the program level the filename TEXT and the file unit number 4 can be associated by the PRIMOS subroutine SRCH$$:

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

Afterwards, other subroutines can access the file by unit number rather than by name, which is faster.

See Chapter 9 for a more thorough discussion of SRCH$$.

As another example, with the aid of the PRIMOS subroutine PRWF$$, the FORTRAN user can bypass formatted I/O and write directly from memory arrays to the file system, as in:

```
    CALL PRWF$$ (K$READ, 1, LOC(TEXT), 36, POS, WORDS, CODE)
```

This subroutine call reads 36 words from the file associated with file unit 1 to the array TEXT. WORDS and CODE are returned values (number of words transferred and error code). POS is the position in the file.

The use of file system subroutines has its advantages and disadvantages. For a PL1G program that does a great deal of I/O, the programmer can save on runtime by calling these subroutines instead of using PL1G I/O statements. However, the program with its subroutine calls is not transportable to a non-Prime machine, and new programmers will not be able to understand or maintain the nonstandard program easily.

General PRIMOS Subroutines

General PRIMOS subroutines include those listed in Chapter 10. (Chapters 9 and 19 through 22 also discuss PRIMOS subroutines with specialized functions.) PRIMOS subroutines are loaded when the FORTRAN library is loaded with LI. They include subroutines for:

- Management of system information
- Global variable management
- Phantom handling
- ACL system management (See Appendix A.)

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. They are available only in R-mode. (See Chapter 11 for the scope and use of this library.)

Applications Library

The Applications library provides users with an easy-to-use library of service routines (Chapter 12). They range from the very simple, which do little more than call a lower-level routine, to relatively high-level functions such as:

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

Subroutines in this library often duplicate the work of subroutines in the File System library, or even call those routines. For example, to delete a file, you may use SRCH$$ or TSRC$$ in the File System library, or you may call DELE$A in the Applications library. If you compare those routines, you will see that DELE$A requires fewer arguments, and is simpler to call. Of course, it may be slightly slower because it makes calls to three subroutines.

Sort Libraries

There are four libraries containing sort subroutines, all presented in Chapter 13:

- VSRTLI subroutines are used to perform most file sorting and merging operations.
- SRTLIB is the R-mode version of VSRTLI.
- VMSORT contains several specialized in-memory sort subroutines and a binary search subroutine.
- MSORTS is the R-mode version of VMSORT.

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. Many of these calls have been outmoded by newer PRIMOS subroutines. The I/O subroutines include:

- Device-independent drivers that route an I/O request to the independent driver, thus allowing the user to maintain device independence. (See Chapter 16.)
- Disk subroutines that perform disk input/output operations. (See Chapter 17.)
- Subroutines that transfer data between a user terminal or paper-tape device and memory. These are helpful, among other things, for using nonprinting characters. (See Chapter 18.)
- Peripheral device routines that control line printers, a printer/plotter, serial and parallel card readers, 7-track, and 9-track tapes. (See Chapter 19.)

Synchronous and Asynchronous Controllers

These subroutines perform the movement of raw data for assigned AMLC or SMLC lines. (See Chapter 20.)

Semaphore-handling Subroutines

PRIMOS supports user applications that have realtime 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 Chapter 21.)

Condition-mechanism Subroutines

The condition mechanism is activated when a program encounters such unexpected occurrences as end of file, illegal address, an attempt to divide by 0, or use of the BREAK key from a terminal.

The condition mechanism's goal is either to repair the problem and restart the program, or to terminate the program in an orderly manner. To achieve this goal, the condition mechanism activates diagnostic or remedial code blocks called on-units.

Users writing in FORTRAN IV, FORTRAN 77, PL1G, or PMA can define their own on-units. However, all these users are automatically protected by PRIMOS system on-units. When an error condition occurs, the condition mechanism looks for on-units within the executing procedure. If it finds none, or if the procedure's on-units call for further help, the condition mechanism searches first through any calling procedures' on-units and then through the system's on-units, activating the first appropriate on-unit it finds.

The system or default on-units, and how to write individualized on-units, are described in Chapter 22 of this guide.

PART II
The Language Interface

3 The BASIC/VM Interface

INTRODUCTION

BASIC/VM has only two types of operand, strings and double-precision (64-bit) floating point. However, when a subroutine is declared in BASIC, several argument types may be declared for the subroutine. The BASIC/VM compiler then handles all conversions of BASIC operands to and from the subroutine argument types.

External functions may not be called from BASIC/VM. However, most functions in this manual may also be called as subroutines.

Table 3-1 summarizes the argument types of FORTRAN and PL1G subroutines that can be called from BASIC/VM, and how to declare these arguments.

To declare a subroutine argument type, use the statement:

```
    SUB FORTRAN sub-name [(type, type...)]
```

The possible types are INT, INT*4, REAL, and REAL*8. The following is a detailed discussion of FORTRAN and PL1G argument types, as well as some generic types, and how they relate to the BASIC/VM data types.

To call a subroutine, use the statement:

```
    CALL sub-name [(arg1, arg2 ...)]
```

Literals may be used as arguments in BASIC/VM subroutine calls.

Table 3-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 3-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

DATA TYPES

INTEGER*2 or FIXED BIN(15)

The INTEGER*2 expected by FORTRAN subroutines is PL1G's FIXED BIN(15), also called just FIXED BIN. It must be declared as INT in BASIC/VM's subroutine declarations. In the BASIC program, the variable or constant to be passed is the normal numeric operand, which is double-precision floating point, and is not explicitly declared.

Sample Program 2 illustrates passing an INTEGER*2 argument.

INTEGER*4 or FIXED BIN(31)

The INTEGER*4 expected by FORTRAN subroutines must be declared as INT*4 in BASIC/VM's subroutine declarations. In the BASIC program, the variable or constant to be passed is the normal numeric operand, which is double-precision floating point, and is not explicitly declared.

Sample Program 3 below illustrates use of an INTEGER*4 argument with the subroutine RNUM$A.

REAL*4

The REAL or REAL*4 argument expected by FORTRAN subroutines must be declared as REAL in BASIC/VM's subroutine declarations. In the BASIC program, the variable or constant to be passed should be used as the normal numeric operand, which is double-precision floating point, and is not explicitly declared.

REAL*8

The REAL*8 argument expected by FORTRAN subroutines must be declared as REAL*8 in BASIC/VM's subroutine declaration. In the BASIC program, the variable or constant to be passed should be the normal numeric operand, which is double-precision floating point, and is not explicitly declared.

Integer Arrays

Integer arrays in FORTRAN may contain either numbers or characters. An integer array should be declared in the BASIC/VM subroutine declaration as INT or INT*4, depending on what the subroutine expects. In the BASIC program, it should be declared either as the array x(y), where x is the variable name and y is the dimension, or as the string X$ with the proper number of characters, again depending on which data type is expected.

Sample Program 1 below illustrates receiving an integer array containing two data types from the subroutine TIMDAT.

> **Caution**
>
> Multidimensional arrays cannot be passed to FORTRAN from other languages, because FORTRAN is the only language to use a column-row format.

ASCII Character (String)

A CHARACTER argument expected by a FORTRAN 77 subroutine should be declared in the BASIC/VM subroutine declaration as INT. In the BASIC program, it should be used as a character string (X$), which is not explicitly declared but must have the number of characters expected by the subroutine.

Sample Program 1 below illustrates receiving a character string from the subroutine TIMDAT.

CHARACTER(n)NONVARYING

This PL1G type, usually declared simply as CHARACTER(n), may be passed as a character string of n characters. The argument should be declared in the BASIC/VM subroutine declaration as INT. In the BASIC program, it should be used as a character string (X$) with the expected number of characters.

String Arrays

String arrays in BASIC cannot be passed as arguments to FORTRAN subroutines.

LOGICAL

LOGICAL or LOGICAL*2 arguments expected by a FORTRAN subroutine should be declared as INT in the BASIC/VM subroutine declaration. In the program, variables or constants to be passed to the subroutine should be used as normal numeric operands (not explicitly declared). They will have a value of 0 (false) or 1 (true).

Sample Program 4 below illustrates accepting a logical argument from the subroutine TEXTO$.

CHARACTER(*)VARYING, POINTER

These arguments expected by FORTRAN or PL1G subroutines cannot be passed from a BASIC/VM program.

BIT(1)

This argument expected by a PL1G subroutine cannot be passed from a BASIC/VM program unless it is declared as BIT(1) ALIGNED. In the latter case, the argument may be treated as an INTEGER*2 whose value is -1 if false.

OTHER THINGS TO KNOW

System Subroutines Not Recognized by BASIC/VM

If a FORTRAN subroutine is in VAPPLB, it may not be recognized by the BASIC/VM compiler. This is because only some of the subroutines from this library have been included in the BASIC/VM compiler so that they may be called by various BASIC/VM commands. Others were omitted because of size considerations. If you make a subroutine call to a routine in VAPPLB (Chapter 12), and it compiles correctly but gives the runtime error message, Entry name xxx not found, then the subroutine is missing from the BASIC/VM compiler and must be installed. Your System Administrator may install more subroutines from VAPPLB (or user-written subroutines) in the BASIC/VM compiler, as explained in the System Administrator's Guide or the BASIC/VM Programmer's Guide.

Sample Program 3 below uses a VAPPLB subroutine, RNUM$A, that is not in the standard BASIC/VM compiler.

SYSCOM Tables

This guide uses names instead of values of certain subroutine arguments. There are three classes of value-names, as described below.

Subroutines in VAPPLB sometimes make reference to codes with names in the format A$xxxx. BASIC cannot accomodate these names, and so the BASIC program must check for the numeric equivalents of these codes. The numeric equivalents are in the table at the end of Chapter 12. They are also listed in the file SYSCOM>A$KEYS.INS.FTN, which can be read or spooled from the terminal.

Some subroutines require keys, which are listed with names in the format K$xxxx. The numeric equivalents of these keys must be read from one of the SYSCOM>KEYS.INS.language files. They are also listed in Appendix C.

Finally, a subroutine may return an error code in the form E$xxxx. The meaning of the numeric error code returned is listed in Appendix D, or may be read from one of the SYSCOM>ERRD.INS.language files.

Sample Program 2 below illustrates use of a numerical equivalent for the key in SYSCOM>KEYS.INS.FTN. Sample Program 3 illustrates the use of A$KEYS.

SAMPLE PROGRAMS

Program 1 — Accepting an Integer Array or Character String

```
10   !THE FOLLOWING PROGRAM ILLUSTRATES A CALL USING A CHARACTER
20   !STRING. IT CALLS THE PRIMOS SUBROUTINE TIMDAT, WHICH RETURNS
30   !AN ARRAY OF MIXED ASCII AND INTEGER FORMAT ELEMENTS.
35   !TO CAPTURE BOTH TYPES IN BASIC, THE SUBROUTINE IS CALLED
40   !TWICE: ONCE WITH ARRAY A AS THE RETURN ARGUMENT, AND THEN
50   !WITH STRING A$ AS THE RETURN ARGUMENT. NOTE ALSO:
60   !              1) VALUES RETURNED START AT A(0).
70   !              2) STORAGE SPACE MUST BE ALLOCATED FOR A$ BEFORE
80   !                 THE CALL.
90   !
110  SUB FORTRAN TIMDAT (INT, INT)
120  DIM A(15)                  REM INTEGER DEFINITION
130  CALL TIMDAT(A(), 28)
140  !
150  A$ = SPA(30)               REM CHARACTER DEFINITION
160  CALL TIMDAT(A$,28)
170  !
180  !BEFORE PRINTING THE RETRIEVED INFORMATION, NOTE THAT THE
190  !FIRST THREE AND LAST RETURNED ARRAY ELEMENTS ARE IN ASCII
200  !FORMAT, SO THEY ARE PRINTED AS RETRIEVED THROUGH A$.
210  !OTHER RETURNED ELEMENTS ARE INTEGERS, SO THEY ARE PRINTED AS
220  !RETRIEVED THROUGH ARRAY A.
230  !
```

```
240  PRINT 'MONTH: ':LEFT(A$,2)
250  PRINT 'DAY: ':MID(A$,3,2)
260  PRINT 'YEAR: ':MID(A$,5,2)
270  PRINT 'TIME IN MINUTES SINCE MIDNIGHT: ':A(3)
275  PRINT 'TIME IN SECONDS: ':A(4)
280  PRINT 'TIME IN TICKS: ' :A(5)
290  PRINT 'LOGIN NAME: ':RIGHT(A$, 25)
300  END
```

To run this program, use the dialog below.

```
OK, BASICV
BASICV REV19.0
>OLD TIMDTB
>RUN


timdtb.basic           THU, DEC 17 1981               10:57:32


TIME IN SECONDS:  0
MONTH:  12
DAY:  17
YEAR:  81
TIME IN MINUTES SINCE MIDNIGHT: 657
TIME IN TICKS: 134
LOGIN NAME:  ANNE
>
```

Program 2 — Using INT*2 and SYSCOM>KEYS

```
10   !THIS SUBROUTINE CALL ILLUSTRATES USE OF THE SYSCOM>KEYS.F
20   !KEYS IN A LANGUAGE THAT CANNOT INVOKE THE SYSCOM TABLE.
30   !
40   PRINT 'BEGINNING OF BASIC PROGRAM'
50   F$ = 'CTRLFL'
60   !
70   !N = K$EXST+K$IUFD+no argument
80   !THEREFORE N = 6 + 0
90   !
100  N = 6
110  L = 6
120  F = 1
130  T = 0
140  SUB FORTRAN SRCH$$(INT, INT, INT, INT, INT, INT)
150  CALL SRCH$$(N,F$,L,F,T,C)
160  PRINT 'CODE IS: ',C
170  END
```

To run this program, use the following dialog. If the file CTRLFL exists, the code displayed will be 0, as explained in Appendix D.

```
OK, BASICV
BASICV REV19.0
>OLD SRCH
>RUN
srch.basic              THU, DEC 17 1981                11:01:23


BEGINNING OF BASIC PROGRAM
CODE IS:                0
>
```

Program 3 — Using an INTEGER*4 Argument

Before this program will work, the subroutine RNUM$A must be installed in BASIC/VM, as explained in the *System Administrator's Guide*. RNUM$A accepts a 32-bit integer as input and checks that it has the correct format.

```
10   !THIS SUBROUTINE CALL ILLUSTRATES USE OF THE INT*4
20   !PARAMETER AND ALSO OF SYSCOM>A$KEYS
30   !
40   PRINT 'BEGINNING OF BASIC PROGRAM'
50   F$ = 'ENTER A NUMBER'
100  L = 14
111  !
112  ! NUMERIC KEY IS A$DEC, EQUAL TO 1
113  N = 1
140  SUB FORTRAN RNUM$A(INT, INT, INT*4, INT)
150  CALL RNUM$A(F$,L,N,V)
160  PRINT 'CODE IS: ',V
170  END
```

Program 4 — Accepting a Logical Argument

Before this program will work, the subroutine TEXTO$ must be installed in BASIC/VM, as explained in the *System Administrator's Guide*.

```
10  REM     A PROGRAM TO CALL SUBROUTINE TEXTO$ TO
20  REM     VERIFY THAT A FILENAME ENTERED BY A USER
30  REM     HAS A VALID FORMAT
40  REM
50  N$ = '                              '
60  SUB FORTRAN TEXTO$(INT, INT, INT, INT)
70  PRINT
80  INPUT "ENTER NAME OF FILE TO BE CREATED: ", N$
90  PRINT
100 L1 = LEN(N$)
```

```
110 CALL TEXTO$(N$, L1, L2, T)
120 IF T = 1 GOTO 210
130 REM
140 REM             LOGICAL T IS FALSE
150 REM
160 PRINT "INVALID NAME - TRY AGAIN"
170 GOTO 80
180 REM
190 REM             LOGICAL T IS TRUE
200 REM
210 PRINT "LENGTH IS", L2
220 PRINT "TRUTH VALUE IS", T
230 PRINT "END OF RUN"
240 END
```

4
The COBOL Interface

INTRODUCTION

To call a subroutine from COBOL, use the format:

```
    CALL 'sub-name' [USING data-name-1 [, data-name-2] ...]
```

The sub-name must be the literal subroutine name enclosed in quotes. The data-names must be described in the DATA division with level-number 01 or 77. Arguments may not be passed to or returned from a subroutine as literals in COBOL. The sample programs below illustrate subroutine calls.

External functions may not be called from COBOL. However, most functions in this book may also be called as subroutines.

DATA TYPES

Table 4-1 summarizes the argument types of FORTRAN and PL1G subroutines that can be called from COBOL. The following is a discussion of FORTRAN and PL1G argument types, as well as some generic types, and how they relate to COBOL data types and structures.

Table 4-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 4-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

INTEGER*2 or FIXED BIN(15)

The INTEGER*2 expected by FORTRAN subroutines is PL1G's FIXED BIN, also called FIXED BIN(15). It must be declared in COBOL programs as COMP, signed or unsigned.

Sample Program 1 illustrates a call to the FORTRAN subroutine TNOUA, which has an INTEGER*2 argument. Sample Program 4 has a call to the PL/I subroutine GV$GET, which expects a FIXED BIN(15) argument.

INTEGER*4, FIXED BIN(31), REAL*4, REAL*8, POINTER

Subroutines that expect arguments of these data types may not be called by COBOL.

BIT(1)

PL1G subroutines that expect arguments of this type may not be called by COBOL, unless the argument is declared in PL1G as BIT(1) ALIGNED. In this case the argument may be passed as COMP, with a value of -1 for false.

Integer Arrays

An integer array in FORTRAN may contain either character or numeric data. The corresponding COBOL operand should be set up as a table of the correct data type to receive the information expected. Sample Program 5 illustrates retrieval of a FORTRAN integer array from the subroutine TIMDAT. Since the array contains both character and numeric data, two COBOL arrays are used.

Multidimensional arrays may not be passed to a FORTRAN subroutine.

ASCII Character String

An ASCII string expected by a FORTRAN subroutine may be declared as PIC 9, PIC X, or PIC A. Sample Program 2 illustrates passing an ASCII string to the subroutine SRCH$$.

LOGICAL

LOGICAL or LOGICAL*2 arguments expected by a FORTRAN subroutine should be declared as COMP in COBOL. The arguments must have a value of 0 (false) or 1 (true).

Sample Program 3 illustrates accepting a logical value from the subroutine TEXTO$.

CHARACTER(*)VARYING

This PL1G data type is implemented as a record structure, with the actual number of characters followed by those characters. The two elements may be represented as follows:

```
 ___________________________________
|0   5  |A   B   C   D   E  |
|___|___|___|___|___|___|___|___|
   COUNT     CHARACTER STRING
```

To declare a comparable structure in COBOL, therefore, requires a two-element record. The record consists of a COMP item containing the actual number of characters, plus a PIC X(n), where _n_ is also the number of characters. The PIC X contains the name to be passed.

Sample Program 4 calls a PL1G subroutine, GV$GET, with two CHAR(*)VAR arguments.

CHARACTER(n)NONVARYING

This PL1G data type, usually declared simply as CHARACTER(n), may be passed as a PIC A or PIC X item of _n_ characters.

OTHER THINGS TO KNOW

Subroutine descriptions in this guide sometimes make reference to codes with names in the format _x$yyyy_. COBOL cannot accomodate these names, and so the COBOL program must check for the numeric equivalents of these codes. There are three categories of these names.

- Some have the format _A$yyyy_. The numeric equivalents are in the table at the end of Chapter 12 on VAPPLB. The equivalents are also listed in the file SYSCOM>A$KEYS.INS.FTN, which can be read or spooled from the terminal.

- Some subroutines require keys, which are listed with names in the format K$yyyy. The numeric equivalents of these keys may be read from one of the SYSCOM>KEYS.INS.x files. They are also listed in Appendix C.

- Finally, a subroutine may return an error code in the form E$yyyy. The meaning of the numeric error code returned is listed in Appendix D, or may be read from one of the SYSCOM>ERRD.INS.x files.

The listings of keys in this guide use decimal numbers, while the files KEYS.INS.X and A$KEYS.INS.X sometimes use octal notation (marked by a colon).

Sample Program 2 shows how COBOL may handle the K$EXST code used by SRCH$$.

SAMPLE PROGRAMS

Program 1 — Using an INTEGER*2 Argument

```
 ID DIVISION.
 PROGRAM-ID. CALC.
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 DISPLAY-TOTAL                  PIC X(8).
 01 INTERMED-TOTAL                 PIC X(8) JUSTIFIED RIGHT.
 01 TOTAL-WORK                     PIC S9(6)V99.
 01 TOTAL-DISPLAY                  PIC -----9.99.
 01 DISPLAY-LINE.
      05 TRANS-CODE                PIC X VALUE 'A'.
     05 TRANS-AMT                  PIC X(8).
 01 TRANS-INTERMED                 PIC X(8) JUSTIFIED RIGHT.
 01 TRANS-WORK                     PIC 9(6)99V99.
 01 ERRBUFF                        PIC XX VALUE '^207'.
 01 COUNTER                        COMP VALUE 1.
*
 PROCEDURE DIVISION.
 000-INITIALIZE.
     DISPLAY '                                                        '.
     DISPLAY 'THIS IS A PROGRAM TO ADD AND SUBTRACT FROM AN INITI
-        'AL TOTAL.'.
     DISPLAY '                                                 '.
     DISPLAY 'WHAT IS INITIAL VALUE OF TOTAL?'.
     DISPLAY '      **  NOTE FORMAT MUST NOT USE DECIMAL POINT.'
     DISPLAY '      **  EX: TO REGISTER $45.25, ENTER 4525.'.
```

```
    ACCEPT DISPLAY-TOTAL.
    UNSTRING DISPLAY-TOTAL DELIMITED BY SPACE INTO
        INTERMED-TOTAL.
    MOVE INTERMED-TOTAL TO TOTAL-WORK.
    DIVIDE 100 INTO TOTAL-WORK.
    DISPLAY 'ENTER AMOUNT< PRECEDED BY :  A FOR ADDITION'.
    DISPLAY '                             S FOR SUBTRACTION'.
    DISPLAY '                             Q FOR QUIT'.
    ACCEPT DISPLAY-LINE.
    PERFORM 013-CONVERT.
    PERFORM 010-PROCESS UNTIL TRANS-CODE = 'Q'.
    PERFORM 030-PRINT-BALANCE.
    STOP RUN.
010-PROCESS.
    IF TRANS-CODE = 'S' PERFORM 011-SUBTRACT,
        ELSE IF TRANS-CODE = 'A' PERFORM 012-ADD,
        ELSE PERFORM 050-PROCESS-ERROR.
    ACCEPT DISPLAY-LINE.
    PERFORM 013-CONVERT.
    EXIT.
011-SUBTRACT.
    SUBTRACT TRANS-WORK FROM TOTAL-WORK.
    MOVE TOTAL-WORK TO TOTAL-DISPLAY.
    DISPLAY 'BALANCE SO FAR:', TOTAL-DISPLAY.
    DISPLAY 'ENTER CODE AND AMOUNT (Q TO QUIT).'.
    EXIT.
012-ADD.
    ADD TRANS-WORK TO TOTAL-WORK.
    MOVE TOTAL-WORK TO TOTAL-DISPLAY.
    DISPLAY 'BALANCE SO FAR:', TOTAL-DISPLAY.
    DISPLAY 'ENTER CODE AND AMOUNT (Q TO QUIT).'.
    EXIT.
013-CONVERT.
    UNSTRING TRANS-AMT DELIMITED BY SPACE INTO TRANS-INTERMED.
    MOVE TRANS-INTERMED TO TRANS-WORK.
    DIVIDE 100 INTO TRANS-WORK.
030-PRINT-BALANCE.
    DISPLAY 'BALANCE IS:'
    DISPLAY TOTAL-DISPLAY.
    EXIT.
 050-PROCESS-ERROR.
     DISPLAY 'FIRST CHARACTER MUST BE A, S, OR Q.'.
     DISPLAY 'ERROR!'.
     CALL 'INOUA' USING ERRBUFF, COUNTER.
     DISPLAY 'MAKE ENTRY AGAIN - Q TO QUIT.'.
```

To compile, load, and run this program, stored as CALC.COBOL, use the following dialog:

```
COBOL CALC

 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI

No Errors, No Warnings, Prlme V-Mode COBOL, Rev. 19.0 <CALC>

OK, SEG -LOAD
[SEG Rev. 19.0]
$ LOAD CALC
$ LI VCOBLB
$ LI
     LOAD COMPLETE
$ Q


OK, SEG CALC

THIS IS A PROGRAM TO ADD AND SUBTRACT FROM AN INITIAL TOTAL.

WHAT IS INITIAL VALUE OF TOTAL?
     **  NOTE FORMAT MUST NOT USE DECIMAL POINT.
     **  EX: TO REGISTER $45.25, ENTER 4525.
4525
ENTER AMOUNT PRECEDED BY :  A FOR ADDITION
                            S FOR SUBTRACTION
                            Q FOR QUIT
A475
BALANCE SO FAR:    50.00
ENTER CODE AND AMOUNT (Q TO QUIT).
M2
FIRST CHARACTER MUST BE A, S, OR Q.
ERROR!
**************************************************
HERE THE BEEP SOUNDS
**************************************************
MAKE ENTRY AGAIN - Q TO QUIT.
A5000
BALANCE SO FAR:   100.00
ENTER CODE AND AMOUNT (Q TO QUIT).
Q
BALANCE IS:
   100.00
OK,
```

Program 2 — Using SYSCOM Keys

Since COBOL cannot use the SYSCOM>KEYS files, the following program uses the equivalent value for K$EXST.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  SRCH-SUB.
*********************************************************
 REMARKS.  THIS PROGRAM CALLS THE SUBROUTINE SRCH$$ TO
     CHECK ON THE EXISTENCE OF A FILE.
*********************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 K-EXST         COMP VALUE 6.
 01 NAME           PIC X(6) VALUE 'CTRLFL'.
 01 NAMELENGTH     COMP VALUE 6.
 01 FUNIT          COMP VALUE 0.
 01 TYPE           COMP VALUE 0.
 01 CODE           COMP.
*
 PROCEDURE DIVISION.
 010-PERFORM-UPDATES.
 020-HOUSEKEEPING.
     CALL 'SRCH$$' USING K-EXST,  NAME, NAMELENGTH, FUNIT, TYPE,
                     CODE.
     DISPLAY 'CODE IS: ', CODE.
```

To compile and load this program, stored as SRCH.COBOL, use the following dialog:

```
OK, COBOL SRCH

 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI

No Errors, No Warnings, Prlme V-Mode COBOL, Rev. 19.0 <SRCH-S>

OK, SEG -LOAD
[SEG rev 19.0]
$ LO SRCH
$ LI VCOBLB
$ LI
LOAD COMPLETE
```

If the file CTRLFL exists, the runtime dialog will be the following (a code of 0 indicates no error):

```
    $ EXEC
    CODE IS: 00000+
    OK,
```

If the file CTRLFL does not exist, the error code will be 15 and the dialog may be the following:

```
    OK, SEG SRCH
    CODE IS: 00015+
    OK,
```

Program 3 -- Using a Logical Value

```
OK, SLIST LOGICAL.COBOL

        IDENTIFICATION DIVISION.
        PROGRAM-ID.  TEXT-OK.
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SOURCE-COMPUTER. PRIME.
        OBJECT-COMPUTER. PRIME.
       *
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01  FILENAME         PIC X(32).
        01  NAMELENGTH       COMP VALUE 32.
        01  TRUELENGTH       COMP.
        01  TEXTOK           COMP.
        01  VALID            PIC XXX VALUE 'NO '.
       *
        PROCEDURE DIVISION.
        010-VERIFICATION.
            DISPLAY 'ENTER NAME OF FILE TO BE CREATED'.
            ACCEPT FILENAME.
            PERFORM 015-NAME-ENTRY UNTIL VALID = 'YES'.
            STOP RUN.
       *
        015-NAME-ENTRY.
            CALL 'TEXTO$' USING FILENAME,NAMELENGTH, TRUELENGTH, TEXTOK.
            DISPLAY 'FILE NAME IS ', TRUELENGTH, ' CHARACTERS LONG'.
            EXHIBIT TEXTOK.
            IF TEXTOK NOT EQUAL 1 DISPLAY 'INVALID FILE NAME-TRY AGAIN',
                ACCEPT FILENAME,
                ELSE MOVE 'YES' TO VALID.
```

This program, stored as LOGICAL.COBOL, may be compiled, loaded, and run with the following dialog:

```
OK, COBOL LOGICAL
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI

No Errors, No Warnings, Prlme V-Mode COBOL, Rev. 19.0 <TEXT-O>

OK, SEG -LOAD
[SEG rev 19.0]
$ LO LOGICAL
$ LI VCOBLB
$ LI
LOAD COMPLETE
$ EXEC
ENTER NAME OF FILE TO BE CREATED
123
FILE NAME IS 00000+ CHARACTERS LONG
TEXTOK= 00000+
INVALID FILE NAME - TRY AGAIN
AAAGH
FILE NAME IS 00005+ CHARACTERS LONG
TEXTOK= 00001+
OK,
```

Program 4 — Using A CHAR(*)VAR Argument

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CHARVAR.
**********************************************************
 REMARKS. THIS PROGRAM CALLS THE SUBROUTINE GV$GET TO
     CHECK THE VALUE OF A GLOBAL VARIABLE BEFORE
     FURTHER PROCESSING. GV$GET HAS CHAR(*)VAR ARGUMENTS.
**********************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
**************************************************
*FOLLOWING ARE THE TWO CHARACTER-VARYING STRUCTURES
**************************************************
```

```
 01 CHAR-VAR.
    05 NCHARS    COMP VALUE 4.
    05 STRING1   PIC X(4) VALUE '.MAX'.
 01 VAR-VALUE.
     05 NCHARS2  COMP VALUE 6.
     05 STRING2  PIC X(6) VALUE SPACES.
*****************************************************
 01 VAR-SIZE     COMP VALUE 6.
 01 CODE         COMP.
*
 PROCEDURE DIVISION.
 020-HOUSEKEEPING.
     CALL 'GV$GET' USING CHAR-VAR, VAR-VALUE, VAR-SIZE, CODE.
     EXHIBIT STRING2.
     EXHIBIT CODE.
     STOP RUN.
```

Before this program is run, global variables must have been defined with dialog such as this:

```
OK, DEFINE_GVAR ANNE>GVARFILE
OK, LIST_VAR
.MIN                                  1
.MAX                                  100
OK,
```

Running the program, stored as CHARVAR.COBOL, would give this result:

```
OK, SEG CHARVAR

STRING2 = 100
CODE= 00000+
OK,
```

Program 5 — Using an Integer Array

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  TIMEDATE.
******************************************************************
 REMARKS. THIS PROGRAM CALLS THE SUBROUTINE TIMDAT, WHICH RETURNS
     AN INTEGER ARRAY. IN COBOL, THIS ARRAY MAY BE RETRIEVED
     EITHER AS A CHARACTER ARRAY (PIC X) OR AS A NUMERIC ARRAY
     (COMP).  AS THE ELEMENTS OF THE ARRAY ARE MIXED CHARACTER
     AND NUMERIC, BOTH FORMS OF ARRAY ARE USED BY COBOL.
******************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. PRIME.
 OBJECT-COMPUTER. PRIME.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 ARRAY.
     05 TABLE    PIC X(30).
***************************************************************
* THIS TABLE IS NOW REDEFINED TWICE, ONCE AS A CHARACTER ARRAY,
* AND ONCE AS A NUMERIC ARRAY:
     05 CHAR-ARRAY REDEFINES TABLE OCCURS 15,  PIC X(2).
     05 NUM-ARRAY REDEFINES TABLE OCCURS 15,   COMP.
***************************************************************
 01  NUMBER     COMP VALUE 15.
*
 PROCEDURE DIVISION.
 010-PERFORM-UPDATES.
 020-HOUSEKEEPING.
     CALL 'TIMDAT' USING ARRAY,  NUMBER.
     DISPLAY 'MONTH IS:  ', CHAR-ARRAY(1).
     DISPLAY 'MINUTES SINCE MIDNIGHT:  ', NUM-ARRAY(4).
     STOP RUN.
```

This program, stored as TIMDTC.COBOL, may be compiled, loaded, and run with the following dialog:

```
OK, COBOL TIMDTC
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI

No Errors, No Warnings, Prlme V-Mode COBOL, Rev. 19.0 <TIMEDA>

OK, SEG -LOAD
[SEG rev 19.0]
$ LO TIMDTC
$ LI VCOBLB
$ LI
LOAD COMPLETE
$ EXEC

MONTH IS:  01
MINUTES SINCE MIDNIGHT:  00564+
OK,
```

5
The FORTRAN Interface

INTRODUCTION

To call a subroutine from FTN or F77, use this format:

```
    CALL sub-name[(identifier [, identifier]...)]
```

where the sub-name is the subroutine name (not in quotes) and the identifiers may be either literals or data-names.

To call a function, use formats such as:

```
    data-name = function-name[(identifier [,identifier]...)]

    IF logical-function[(identifier [, identifier])]...  GO TO 100
```

The sample programs below illustrate subroutine and function calls from both FTN and F77.

DATA TYPES

Table 5-1 summarizes the argument types of FORTRAN and PL1G subroutines and functions that can be called from FORTRAN IV (FTN) and FORTRAN 77 (F77).

Table 5-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 5-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

Most older subroutines are written in FORTRAN IV (FTN), so the data types used by FTN are used as the norm in this chapter. The following discussion concentrates on how to make any conversions necessary for F77, and how to handle in FORTRAN the data types that are expected by PL1G subroutines.

The Data Type INTEGER

Beware of using integer arguments that are not explicitly declared as INTEGER*2 or INTEGER*4. By default, FORTRAN IV stores such arguments as INTEGER*2, while FORTRAN 77 stores them as INTEGER*4. This is true of constants as well as variables. Thus it is safer to pass all numeric arguments as explicitly declared variables.

You may also avoid the contradiction by using the -INTS (short integer) option when compiling an F77 program. In this case, you must remember to add the option every time you recompile.

Note

> In this guide, if an argument is described as integer, it should be treated as INTEGER*2.

INTEGER, INTEGER*2, FIXED BIN(15)

The first two names are the same data type in FTN. The equivalent for F77 is INTEGER*2, or INTEGER if the program is compiled with the -INTS (short integer) option. FIXED BIN and FIXED BIN(15) are equivalent in PL1G. The data is stored as a 16-bit half-word.

INTEGER*4, FIXED BIN(31)

These are the same data type. The equivalent for F77 is INTEGER*4, or INTEGER if the program is compiled without the -INTS option. The data type is stored as a 32-bit word. (If FORTRAN IV is compiled with the -INTL option, INTEGER may be used as INTEGER*4.)

LOGICAL

The equivalent for F77 is LOGICAL*2. The data type is stored as a 16-bit half-word. Sample Program 3 illustrates a call with LOGICAL arguments.

BIT(1)

PL1G programs using this data type may not be called from FORTRAN unless the argument is declared in PL1G as BIT(1) ALIGNED. Then it may be used in FORTRAN as an INTEGER*2 and will have a value of -1 if false.

REAL, REAL*4, REAL*8

REAL*4 is a single-precision (32-bit) floating-point number. REAL*8 is a double-precision (64-bit) floating-point number. REAL is equivalent to REAL*4 in both FORTRANs.

ASCII Character Data

A FTN subroutine that expects an ASCII string will accept INTEGER*2 or, from F77, CHARACTER*n.

CHARACTER(*)VARYING

This P11G data type is implemented as a record structure, with the actual number of characters followed by those characters. The two elements may be represented in the following way:

```
 ________________________________
|0    5  |A   B   C   D   E  |
|________|___|___|___|___|___|
  COUNT     CHARACTER STRING
```

To declare a comparable structure in FORTRAN, therefore, requires a two-element record. The record consists of an INTEGER*2 item containing the actual number of characters, plus a field for the character string. This field may be CHARACTER*n in F77, or INTEGER*2 in FTN, and should contain the characters to be passed.

A good way to set up such a record is by using the EQUIVALENCE statement to assign different parts of a data name to different items. Consider the following FTN example:

```
    INTEGER*2 STRING(10), LENGTH
    INTEGER*2 VARSTRING(11)
    EQUIVALENCE (LENGTH, VARSTRING(1))
    EQUIVALENCE (STRING(1), VARSTRING(2))
```

This code sets up a record that may be represented this way.

```
 ____________________________________________________
|           |                                        |
|  LENGTH   |   STRING                               |
|___________|________________________________________|
<-----------VARSTRING ------------------------------->
```

The data names may then be given values and the PL1G subroutine may be called with this code:

```
STRING(1) = 'MY'
STRING(2) = 'FI'
STRING(3) = 'LE'
LENGTH = 6
CALL PL1G-SUB(VARSTRING)
CALL EXIT
```

The value 6 is assigned to LENGTH because six characters are actually assigned to STRING.

In F77 the code can be a little simpler because all of STRING can be assigned at once:

```
INTEGER*2 LENGTH, VARSTRING(11)
CHARACTER*20 STRING
EQUIVALENCE(LENGTH, VARSTRING(1))
EQUIVALENCE(VARSTRING(2), STRING)
STRING(1:6) = 'MYFILE'
LENGTH = 6
CALL PL1G-PROG(VARSTRING)
CALL EXIT
```

Sample Programs 4 and 5 below call a PL1G routine, GV$GET, with two CHAR(*)VAR arguments.

CHARACTER(n)NONVARYING

This PL1G data type, usually declared simply as CHARACTER(n), may be represented in FORTRAN 77 simply as CHARACTER*n. In FORTRAN IV it should be represented as an integer array and the data name should be followed by the number of words (one-half the value of the (n) in CHARACTER(n), rounded). An example is INTEGER*2 STR(N/2 + 5).

Array

When a FORTRAN subroutine expects an array, an ASCII character array (data type INTEGER*2 or CHARACTER*n) may be used. Sample Program 2 shows how to use an integer array returned to FTN.

Note

CHARACTER*n does not necessarily allocate data on word boundaries. Thus not all routines called from VAPPLB will work with this data type.

POINTER

PL1G subroutines that expect this data type should not generally be called from FORTRAN. For experienced programmers, the expression LOC(name) may be passed to a subroutine that expects a pointer. See note 6 to Table 5-1.

USING SYSCOM TABLES

In this guide, numeric values are often represented by a name in the form y$xxxx, where y and x are characters of the alphabet. The code name or key name may be used instead of a numeric value. There are three files in the SYSCOM UFD that are of use in handling these names. SYSCOM>A$KEYS.INS.FTN, SYSCOM>KEYS.INS.FTN, and SYSCOM>ERRD.INS.FTN contain keys that should be used instead of numeric values for codes.

To use these key names in a FORTRAN program, use $INSERT SYSCOM>xxx at the beginning of each program module with the declarations of other data names. The file A$KEYS.INS.FTN also contains declarations for all of the subroutines in VAPPLB or APPLIB.

Sample Program 1 illustrates use of the keys from SYSCOM files.

SAMPLE FTN (FORTRAN IV) PROGRAMS

Program 1 — Using SYSCOM Keys

```
OK, SLIST SRCH.FTN

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C   THIS PROGRAM CALLS THE SUBROUTINE SRCH$$ TO CHECK      C
C   ON THE EXISTENCE OF A FILE.  THE PROGRAM ALSO USES     C
C   THE SYSCOM FILES FOR KEY CODES.                        C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      INTEGER*2 CODE, TYPE,FUNIT, POS
$INSERT SYSCOM>KEYS.INS.FTN
      WRITE(1,100)
      FUNIT = 4
      POS = 0
      CALL SRCH$$ (K$EXST+K$IUFD, 'CTRL', FUNIT, POS, TYPE, CODE)
      IF (CODE .NE. 0) WRITE(1, 200)CODE
      WRITE(1,300)
```

```
          CALL EXIT
    100   FORMAT('THIS IS FORTRAN')
    200   FORMAT('CODE IS ', I2)
    300   FORMAT('END OF RUN')
          END
```

This program, stored as SRCH.FTN, may be compiled, loaded, and run in R-mode with the following dialog. If the file CTRL does not exist, the return code will be 15 (E$FNTF).

```
    OK, FTN SRCH
    0000 ERRORS [<.MAIN.>FTN-REV18.4]

    OK, LOAD
    [LOAD rev 19.0.1]
    $ LO SRCH
    $ LI
    LOAD COMPLETE
    $ SA
    $ EXEC

    THIS IS FORTRAN
    CODE IS 15
    END OF RUN
    OK,
    OK,
```

Program 2 — Integer Arrays

The subroutine TIMDAT returns an array containing both ASCII characters and integers. The following program handles these two types, both in STRING, differently by means of the EQUIVALENCE statement. It prints STRING(13) through STRING(15) as NAME, in A format, and prints STRING(1) through STRING(6) as TIME, in I format.

```
          INTEGER*2 STRING(28)
          INTEGER*2 NUM, DATE(3)
          INTEGER*2 TIME, TIME1, TIME2, NAME(16)
          EQUIVALENCE (STRING(1), DATE)
          EQUIVALENCE (STRING(4), TIME)
          EQUIVALENCE (STRING(5), TIME1)
          EQUIVALENCE (STRING(6), TIME2)
          EQUIVALENCE (STRING(13), NAME)
          NUM = 28
          CALL TIMDAT(STRING, NUM)
          WRITE (1, 300) DATE
          WRITE (1,400)
          WRITE(1,200) TIME, TIME1, TIME2
          WRITE(1,150) NAME
    200   FORMAT (I6, I6, I6)
    150   FORMAT ('USER IS ',3A2)
    300   FORMAT ('DATE IS ',3A2)
```

```
400   FORMAT ('TIME SINCE MIDNIGHT IN MINUTES+SECONDS+TICKS:  ')
      CALL EXIT
      END
```

This program, stored as TIMDTF.FTN, may be compiled, loaded, and run in V-mode as follows:

```
OK, FTN TIMDTF -64V
0000 ERRORS [<.MAIN.>FTN-REV18.4]
OK, SEG -LOAD
[SEG rev 19.0.1]
$ LO TIMDTF
$ LI
LOAD COMPLETE
$ EXEC

DATE IS 121781
TIME SINCE MIDNIGHT IN MINUTES + SECONDS + TICKS:
   692    57    75
USER IS ANNE
OK,
```

Program 3 — Using a Logical Function

This program calls DELE$A to delete a file and return a truth value according to its success.

```
OK, SLIST LOGICAL.FTN

      INTEGER*2 LENGTH
      LOGICAL*2 DELE$A
      LENGTH = 6
      IF (DELE$A('CTRLFL', LENGTH)) GOTO 50
      WRITE(1,200)
      CALL EXIT
50    WRITE(1,100)
      CALL EXIT
100   FORMAT ('DELETE WAS SUCCESSFUL')
200   FORMAT ('NO GO')
      END
```

This program may be compiled, loaded, and run with the following dialog:

```
OK, FTN LOGICAL -64V
0000 ERRORS [<.MAIN.>FTN-REV18.4]
OK, SEG -LOAD
[SEG rev 19.0.1]
$ LO LOGICAL
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

DELETE WAS SUCCESSFUL
```

If this program is run when CTRLFL does not exist, the following will happen:

```
OK, SEG LOGICAL
Not found. CTRLFL (DELE$A)
NO GO
OK,
```

Program 4 — Using CHAR(*)VARYING Arguments

This program calls GV$GET to return the value of a PRIMOS (CPL) global variable.

```
OK, SLIST GVAR.FTN

      INTEGER*2 CODE
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C The next 7 lines define two   CHAR*VARs.   C
      INTEGER*2 STR1(10), STR2(10), LEN1, LEN2
      INTEGER*2 VARNAM(11)
      INTEGER*2 VARVAL(11)
      EQUIVALENCE(LEN1, VARNAM(1))
      EQUIVALENCE(LEN2, VARVAL(1))
      EQUIVALENCE(VARNAM(2), STR1(1))
      EQUIVALENCE(VARVAL(2), STR2(1))
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      STR1(1) = '.M'
      STR1(2) = 'AX'
      LEN1 = 4
      CALL GV$GET(VARNAM,VARVAL, 20, CODE)
      WRITE(1,100) CODE
      WRITE(1,200)STR2
100   FORMAT('CODE IS',I3)
200   FORMAT('.MAX IS ', 10A2)
      CALL EXIT
      END
```

This program may be compiled, loaded, and run with the following dialog, providing that the global variable file has previously been established as explained in the CPL User's Guide.

```
DEFINE_GVAR GVARFILE -CREATE
OK, SET_VAR .MAX = 100
OK,
```

Note

This program may only be compiled in V-mode, because it calls a V-mode subroutine.

```
OK, FTN GVAR -64V
0000 ERRORS [<.MAIN.>FTN-REV18.4]

OK, SEG -LOAD
[SEG rev 19.0.1]
$ LO GVAR
$ LI
LOAD COMPLETE
$ EXEC

CODE IS  0
.MAX IS 100
OK,
```

SAMPLE F77 (FORTRAN 77) PROGRAM

The sample programs above may be used unchanged with F77 if the -INTS compile option is used. These programs demonstrate the use of integers, characters, and codes from SYSCOM files included with $INSERT. The following program may be used only with F77.

Program 5 — Using CHAR(*)VARYING with F77

```
OK, SLIST GVAR.F77

      INTEGER*2 CODE, LEN1, LEN2, VARLEN
      CHARACTER*20 STR1, STR2
      INTEGER*2 VARNAM(11)
      INTEGER*2 VARVAL(11)
      EQUIVALENCE(LEN1, VARNAM(1))
      EQUIVALENCE(LEN2, VARVAL(1))
      EQUIVALENCE(VARNAM(2), STR1)
      EQUIVALENCE(VARVAL(2), STR2)
      LEN1 = 4
      VARLEN = 20
      STR1 = '.MAX'
      CALL GV$GET(VARNAM,VARVAL, VARLEN, CODE)
      WRITE(1,100) CODE
      WRITE(1,200)STR2
100   FORMAT('CODE IS',I4)
200   FORMAT('.MAX IS ', A20)
      CALL EXIT
      END
```

This program, stored as GVAR.F77, may be compiled, loaded, and run with the following dialog:

```
    OK, F77 GVAR
    [FORTRAN 77 19.0]
    0000 ERRORS [<.MAIN.> F77-REV19.0]

    OK, DEFINE_GVAR ANNE>GVARFILE

    OK, SEG -LOAD
    [SEG rev 19.0.1]
    $ LO GVAR
    $ LI
    LOAD COMPLETE
    $ EXEC

    CODE IS   0
    .MAX IS 100
    OK,
```

SAMPLE FILE SYSTEM PROGRAMS

This section contains sample programs illustrating the use of the file system subroutines in Chapter 9. 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

The programs also illustrate the use of PRWF$$, SGDR$$, and SRCH$$ to read and write to a file.

Program 6 — Writing a SAM File

```
OK, SLIST SAMWRITE.FTN

C SAMWRT  BIN    PROGRAM TO WRITE A SAM DATA FILE
C THE FILE IS 1000 WORDS LONG WRITTEN FROM ARRAY BUFF
C RESTRICTIONS: SAMFIL SHOULD NOT EXIST BEFORE PROGRAM IS RUN
C
      INTEGER*2 FUNIT1  /* FILE UNIT TO BE USED
      INTEGER*2 SAMFIL  /* FILE TYPE FOR SAM FILE
      INTEGER*2 BUFLNG  /* BUFFER LENGTH
      PARAMETER (SAMFIL=0, BUFLNG=1000)
      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
$INSERT SYSCOM>KEYS.INS.FTN
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
      CALL SRCH$$(K$WRIT+K$GETU+K$NSAM,'SAMFIL',6,FUNIT1,TYPE,
     X    CODE)
      IF (CODE.NE.0) GO TO 9010
      IF (TYPE. NE. SAMFIL) GO TO 9000   /* ERROR
C
C WRITE 1000 WORDS FROM BUFF INTO THE NEW DATA FILE
C
      CALL PRWF$$(K$WRIT,FUNIT1,LOC(BUFF),BUFLNG,INTL(0),NMREAD,
     X    CODE)
      IF (CODE.NE.0) GO TO 9010
C
C CLOSE FILE. THIS RELEASES UNIT FUNIT1 FOR REUSE AND ASSURES
C ALL FILE BUFFERS HAVE BEEN WRITTEN TO DISK.
C NOTE PRIMOS WILL NOT AUTOMATICALLY CLOSE FILES ON 'CALL EXIT'.
C
9000  CALL SRCH$$(K$CLOS, 0, 0, FUNIT1, 0, CODE)
      IF (CODE.NE.0) GO TO 9010
9010  WRITE(1,9012)
9012  FORMAT('ERROR!')
C
C RETURN TO PRIMOS
C
      CALL EXIT
      END
```

This program, stored as SAMWRITE.FTN, may be compiled, loaded, and run with the following dialog. It will create the data file SAMFIL.

```
    OK, FTN SAMWRITE
    0000 ERRORS [<.MAIN.>FTN-REV18.4]
    OK, LOAD
    [LOAD rev 19.0.1]
    $ LO SAMWRITE
    $ LI
    LOAD COMPLETE
    $ SA
    $ EXEC
    OK,
```

Program 7 — Writing a DAM File

```
OK, SLIST DAMWRITE.FTN
C DAMWRT  BIN    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$$ IN CREATING THE FILE
C
C RESTRICTION: DAMFIL SHOULD NOT EXIST BEFORE RUNNING PROGRAM
C
      INTEGER*2 FUNIT1  /* FILE UNIT TO BE USED
      INTEGER*2 DAMFIL  /* FILE TYPE OF DAM DATA FILE
      INTEGER*2 BUFLNG  /* DATA BUFFER LENGTH IN WORDS
C
      PARAMETER (DAMFIL=1, BUFLNG=1000)
C
      INTEGER*2 BUFF(BUFLNG)  /* DATA BUFFER
      INTEGER*2 TYPE    /* FILE TYPE RETURNED BY SRCH$$
      INTEGER*2 NMREAD  /* NUMBER WORDS READ OR WRITTEN BY PRWF$$
      INTEGER*2 CODE    /* ERROR CODE RETURNED FROM FILE SYSTEM
      INTEGER*2 I
C
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
C INITIALIZE BUFFER
C
      DO 10 I = 1, BUFLNG
        BUFF(I) = I
10    CONTINUE
C
C ASSURE THAT THE FILE 'DAMFIL' DOES NOT ALREADY EXIST
C
      CALL SRCH$$(K$EXST+K$IUFD,'DAMFIL',6,FUNIT1,TYPE,CODE)
      IF (CODE .NE. E$FNTF) GO TO 9000  /* FILE ALREADY EXISTS
C
C OPEN A NEW DAM FILE CALLED 'DAMFIL' IN THE CURRENT
C UFD FOR WRITING ON FILE UNIT FUNIT1
```

```
C
      CALL SRCH$$(K$WRIT+K$GETU+K$NDAM,'DAMFIL',6,FUNIT1,TYPE,
     X   CODE)
      IF (CODE.NE.0) GO TO 9010
      IF (TYPE .NE. DAMFIL) STOP   /* WILL NEVER STOP
C
C WRITE THE BUFFER INTO THE FILE
C
      CALL PRWF$$(K$WRIT,FUNIT1,LOC(BUFF),BUFLNG,INTL(0),NMREAD,
     X   CODE)
      IF (CODE.NE.0) GO TO 9010
C
C K$CLOS THE FILE AND EXIT
C
9000  CALL SRCH$$(K$CLOS, 0, 0, FUNIT1, TYPE, CODE)
      IF (CODE.NE.0) GO TO 9010
      CALL EXIT
C
9010  CALL ERRPR$(K$NRTN,CODE,0,0,0,0)
      END
```

This program, stored as DAMWRITE.FTN, may be compiled, loaded, and run with the following dialog. A data file called DAMFIL will be created.

```
OK, FTN DAMWRITE
0000 ERRORS [<.MAIN.>FTN-REV18.4]

OK, LOAD
[LOAD rev 19.0.1]
$ LO DAMWRITE
$ LI
LOAD COMPLETE
$ SA
$ EX
OK,
```

Program 8 — Reading a SAM or DAM File

```
OK, SLIST SAMREAD.FTN

C REDFIL  BIN    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
      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=2, 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$$
C
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
C INITIALIZE AND GET FILE NAME FROM TERMINAL
C
      LARGST = -32767  /* LARGEST UNSIGNED INTEGER
10    WRITE(1,1000)  /* FORTRAN UNIT 1 IS TERMINAL
1000  FORMAT ('TYPE FILE NAME')
C
      READ(1,1010) (FNAME(I), I=1,16)
1010  FORMAT (16A2)
C
C OPEN FNAME IN CURRENTLY ATTACHED UFD FOR READING ON FILE UNIT 1
C (NOT THE SAME AS FORTRAN UNIT 1). CHECK FOR ERRORS.
C NOTE THAT THE NAME NEED NOT ACTUALLY BE 32 CHARACTERS LONG AS
C TRAILING BLANKS ARE IGNORED.
C
      CALL SRCH$$(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$EOF) GO TO 31  /* END-OF-FILE
      IF (CODE .NE. 0) GO TO 9010  /* SOME OTHER ERROR
      WRITE(1,3)BUFF(I)
3     FORMAT(I6)
31    DO 40 I= 1, NMREAD    /* FOR EACH WORD ACTUALLY READ
         IF ((LARGST.LE.0).AND.(BUFF(I).GE.0)) LARGST = BUFF(I)
         IF (LARGST .LT. BUFF(I)) LARGST = BUFF(I)
40    CONTINUE
      IF (CODE .NE. E$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
```

This program may be compiled, loaded, and run to read the file SAMFIL created by the first program in this section with the following dialog:

```
OK, FTN SAMREAD
0000 ERRORS [<.MAIN.>FTN-REV18.4]

OK, LOAD
[LOAD rev 19.0.1]
$ LO SAMREAD
$ LI
LOAD COMPLETE
$ SA
$ EXEC
TYPE FILE NAME
SAMFIL
```

```
          16
         200
         300
         400
         500
         600
         700
         800
         900
        1000
    LARGEST INTEGER IN FILE IS   1000
    OK,
```

Program 9 — Creating a Segment Directory

```
OK, SLIST SEGWRITE.FTN
C CRTSEG  BIN   CREATE A SEGMENT DIRECTORY
C                AND WRITE DATA FILE IN IT
C
C RESTRICTIONS: SEGDIR SHOULD NOT EXIST BEFORE PROGRAM IS RUN
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=10, 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.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
C INITIALIZE DATA BUFFER CONTENTS
C
      DO 10 I= 1, BUFLNG
        BUFF(I) = I
10    CONTINUE
C
C OPEN A NEW SAM SEGMENT DIRECTORY CALLED 'SAMDIR' IN CURRENTLY
C ATTACHED UFD FOR READING AND WRITING ON FILE UNIT SGUNIT.
C NOTE: SEGDIRS OPEN FOR WRITE ONLY WILL NOT BE HANDLED CORRECTLY
C
      CALL SRCH$$(K$RDWR+K$NSGS+K$IUFD,'SAMDIR',6,SGUNIT,TYPE,
     X    CODE)
      IF (CODE.NE.0) GO TO 9500
      IF (TYPE.NE.SAMSEG) GO TO 9500  /* ERROR--MUST HAVE EXISTED
```

```
C
C ENTER A NEW SAM DATA FILE (I.E. OPEN SAM DATA FILE FOR WRITING)
C IN THE SEGMENT DIRECTORY JUST CREATED. THE NEW DATA FILE
C WILL BE ENTRY 0 IN THE SEGMENT DIRECTORY.
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,SGUNIT,0,FUNIT,TYPE,CODE)
      IF (CODE.NE.0) GO TO 9500
C
C WRITE THE DATA BUFFER INTO THE SAM FILE JUST CREATED.
C K$CLOS THE DATA FILE.
C
      CALL PRWF$$(K$WRIT,FUNIT,LOC(BUFF),BUFLNG,INTL(0),NMREAD,
     X    CODE)
      IF (CODE.NE.0) GO TO 9500
      CALL SRCH$$(K$CLOS, 0, 0, FUNIT, 0, CODE)
      IF (CODE.NE.0) GO TO 9500
C
C REPLACE BUFF WITH NEW DATA
C
      DO 20 I= 1, BUFLNG
        BUFF(I) = I * 10
20    CONTINUE
C
C OPEN A DIFFERENT NEW SAM DATA FILE ON FUNIT FOR WRITING
C (I.E. ENTER ANOTHER FILE IN SEGMENT DIRECTORY). THIS IS DONE
C IN TWO STEPS. FIRST THE FILE POINTER OF THE SEGMENT DIR UNIT IS
C POSITIONED TO THE ENTRY NUMBER DESIRED. THE SRCH$$ IS
C CALLED AS ABOVE.
C
      CALL SGDR$$(K$SPOS,SGUNIT, 1, I, CODE)
      IF (CODE.NE.0) GO TO 9500
      IF (I .NE. -1) GO TO 9500  /* ERROR EXIT
C
C NOTE THAT THE SEGMENT DIRECTORY OPEN ON SGUNIT HAS ONLY 1 ENTRY
C (ENTRY 0) AT THIS TIME. THUS, POSITIONING TO ENTRY 1
C WILL POSITION TO END-OF-FILE (NOT BEYOND) AND THE FOLLOWING
C CALL TO SRCH$$ WILL CAUSE THE SEGMENT DIRECTORY TO BE EXTENDED
C IN LENGTH BY ONE ENTRY.
C
      CALL SRCH$$(K$WRIT+K$NSAM+K$ISEG,SGUNIT,0,FUNIT,TYPE,CODE)
      IF (CODE.NE.0) GO TO 9500
C
C WRITE DATA INTO THE SAM FILE   THE K$CLOS THE FILE
C
      CALL PRWF$$(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, 0, CODE)
      IF (CODE.NE.0) GO TO 9500
C
C REPLACE THE BUFFER WITH NEW DATA
C
      DO 30 I= 1, BUFLNG
        BUFF(I) = I * 100
```

```
30    CONTINUE
C
C MAKE THE SEGMENT DIRECTORY ITSELF LARGE ENOUGH TO CONTAIN
C 10 ENTRIES. PLACE A SAM FILE IN THE 10TH ENTRY.
C
      CALL SGDR$$(K$MSIZ, SGUNIT, 10, 0, CODE)
      IF (CODE.NE.0) GO TO 9500
C
C THE FILE POINTER ASSOCIATED WITH SGUNIT IS NOW AT END-OF-FILE.
C A CALL TO SRCH$$ 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, 9, 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
```

This program, stored as SEGWRITE.FTN, may be compiled, loaded, and run with the following dialog. It will create an empty segmented file called SAMDIR.

```
OK, FTN SEGWRITE
0000 ERRORS [<.MAIN.>FTN-REV18.4]
OK, LOAD
[LOAD rev 19.0.1]
$ LO SEGWRITE
$ LI
```

```
LOAD COMPLETE
$ SA
$ EXEC
OK,
```

Program 10 — Reading a Logical Record from a File

```
OK, SLIST LOGICREAD.FTN

C RDLREC  BIN   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.
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 0 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
      INTEGER*2 FUNIT1  /* PRIMOS FILE UNIT USED FOR DATA FILE
      INTEGER*2 BUFLNG  /* LENGTH OF DATA BUFFER
C
      PARAMETER (FUNIT1=2, BUFLNG=1000)
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
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
C ASK FOR FILENAME
C
10    WRITE(1,1000)   /* FORTRAN UNIT 1 IS TTY
```

```
1000  FORMAT ('TYPE FILE NAME')
C
C READ FILE NAME
C
      READ(1,1010) (FNAME(I),I=1,16)
1010  FORMAT (16A2)
C
C OPEN FNAME IN CURRENT UFD FOR READING ON FILE UNIT FUNIT2
C
      CALL SRCH$$(K$READ+K$IUFD, FNAME, 32, FUNIT1, TYPE, CODE)
      IF (CODE.NE.0) GO TO 2000
C
C ASK FOR LOGICAL RECORD SIZE
C
20    WRITE(1,1020)
1020  FORMAT ('TYPE RECORD SIZE')
      READ(1,1030) RECSIZ
1030  FORMAT (I6)
      IF (RECSIZ .GE. 1 .AND. RECSIZ .LE. BUFLNG) GO TO 30
      WRITE(1,1040)
1040  FORMAT ('BAD RECORD SIZE')
      GO TO 20
C
C ASK FOR RECORD NUMBER. FIRST RECORD IS NUMBERED 0
C
30    WRITE(1,1050)
1050  FORMAT ('TYPE RECORD NUMBER')
      READ (1,1030) RECNUM
      IF (RECNUM .GE. 0) GO TO 35
      WRITE(1,1051)
1051  FORMAT ('BAD RECORD NUMBER')
      GO TO 30
C
C CALCULATE THE 32-BIT WORD NUMBER OF THE FIRST WORD IN THE
C DESIRED RECORD. NOTE THAT IF  RECSIZ AND RECNUM ARE BOTH
C POSITIVE 16BIT NUMBERS, THE 32BIT WORD NUMBER MUST ALSO BE
C POSITIVE.
C
C POSITIONING MAY BE DONE TO AN ABSOLUTE WORD NUMBER OR RELATIVE
C TO THE CURRENT POSITION. SINCE A JUST OPENED FILE IS ALWAYS
C POSITIONED TO TOP-OF-FILE AND THE CALCULATED WORD NUMBER WILL
C NEVER BE NEGATIVE, THE ARGUMENT FOR POSITION TO PRWF$$ 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 100  /* 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$$(K$READ+K$PRER,FUNIT1,LOC(BUFF),RECSIZ,POSITN,
     X                                         NMREAD, CODE)
      GO TO 200  /* SKIP OVER ABSOLUTE POSITION EXAMPLE
C
C RECORD IS MORE THAN 32767 WORDS FROM THE BEGINNING OF FILE, USE
C ABSOLUTE POSITIONING.
C
C WHEN ABSOLUTE POSITIONING IS USED, POSITION ARGUMENT (POSITN)
C IS CONSIDERED TO BE AN SIGNED 32-BIT INTEGER.
C NOTE THAT THE E$BOF ERROR (BEGINNING OF FILE) CAN OCCUR.
C
100   CALL PRWF$$(K$READ+K$PREA,FUNIT1,LOC(BUFF),RECSIZ,POSITN,
     X                                         NMREAD, CODE)
C
200   IF (CODE .NE. 0) GO TO 300  /* ERROR DETECTED
C
C HAVE READ RECORD, NOW DISPLAY 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
2000  CALL ERRPR$(K$NRTN,CODE,0,0,0,0)
      END
```

This program, compiled, loaded, and stored as LOGICREAD.SAVE, may be run with the following dialog:

```
    OK, R LOGICREAD
    TYPE FILE NAME
    SAMFIL
    TYPE RECORD SIZE
    1
    TYPE RECORD NUMBER
    0
    RECORD      0 CONTAINS      1 ENTRIES AS FOLLOWS
          1
    OK, R LOGICREAD
    TYPE FILE NAME
    SAMFIL
    TYPE RECORD SIZE
    1
    TYPE RECORD NUMBER
    8
    RECORD      8 CONTAINS      1 ENTRIES AS FOLLOWS
          9
    OK,
```

Program 11 — Reading a File in a Segment Directory

```
OK, SLIST SEGREAD.FTN

C REDSEG  BIN   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 0. THE FIRST WORD IN THE FILE IS
C WORD NUMBER 0.
C
C RESTRICTIONS:
C   1. THE SEGMENT DIRECTORY FILE MUST EXIST
C   2. THE FILE NUMBER MUST BE BETWEEN 0 AND 32767
C   3. THE FILE MUST BE IN THE SEGMENT DIRECTORY
C   4. THE WORD NUMBER MUST BE BETWEEN 0 AND 32767
C   5. THE WORD MUST BE IN THE FILE.
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.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
C
C
C ASSURE FILE UNITS TO BE USED ARE K$CLOSD
C ASK FOR AND READ SEGMENT DIRECTORY NAME FROM TERMINAL
C
10    CALL SRCH$$(K$CLOS, 0, 0, SGUNIT, 0, CODE)
      IF (CODE.NE.0) GO TO 100
      CALL SRCH$$(K$CLOS, 0, 0, FUNIT, 0, CODE)
      IF (CODE.NE.0) GO TO 100
      WRITE(1,1000)
1000  FORMAT ('TYPE SEGMENT DIRECTORY NAME')
      READ (1,1010) (SEGDIR(I), I=1,16)
1010  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.0) GO TO 100
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 20
      IF (TYPE .EQ. DAMSEG) GO TO 20
C
C NOT A SEGMENT DIRECTORY - TRY AGAIN
C
      WRITE(1,1020)
1020  FORMAT('FILE IS NOT A SEGMENT DIRECTORY')
      GO TO 10
C
C ASK FOR FILE (ENTRY) NUMBER IN SEGMENT DIRECTORY
C
20    WRITE(1,1030)
1030  FORMAT ('TYPE FILE NUMBER')
      READ (1,1040) FILNUM
1040  FORMAT (I6)
      IF (FILNUM .LT. 0) GO TO 20
C
C ASK FOR WORD NUMBER IN DATA FILE TO READ
C
30    WRITE(1,1035)
1035  FORMAT ('TYPE WORD NUMBER')
      READ (1,1040) WRDNUM
      IF (WRDNUM .LT. 0) GO TO 30
```

```
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$$(K$SPOS, SGUNIT, FILNUM, TYPE, CODE)
      IF (CODE .EQ. E$EOF) CODE = E$FNTS  /* FILE NOT FOUND
      IF (CODE .NE. 0) GO TO 100
C
C OPEN FILE IN SEGMENT DIRECTORY FOR READING
C
      CALL SRCH$$(K$READ+K$ISEG,SGUNIT,0,FUNIT,TYPE,CODE)
      IF (CODE .NE. 0) GO TO 100
C
C PRINT THE WORD, K$CLOS THE FILES, AND RETURN TO PRIMOS
C
      WRITE(1,1050) WRDNUM,FILNUM,(SEGDIR(I), I= 1,16),BUFF
1050  FORMAT ('WORD',I6,' OF FILE (',I6,') IN ',16A2,
     X  'CONTAINS',I6)
50    CALL SRCH$$(K$CLOS, 0, 0, FUNIT, 0, CODE)
      CALL SRCH$$(K$CLOS, 0, 0, SGUNIT, 0, CODE)
      CALL EXIT
      GO TO 10  /* START COMMAND RESTARTS PROGRAM
C
C COMMON ERROR HANDLER
C
100   IF (CODE.EQ.E$FNTS) GO TO 110 /* FILE NOT FOUND IN SEGDIR
      IF (CODE .EQ. E$EOF ) GO TO 120   /* END-OF-FILE
      CALL ERRPR$(K$IRTN,CODE,0,0,'REDSEG',6) /* PRINT ERROR MSG
      GO TO 50   /* K$CLOS FILES   EXIT
C
C FILE NOT FOUND IN SEGMENT DIRECTORY
C LET THE USER TRY AGAIN
C
110   WRITE(1,1060) FILNUM, (SEGDIR(I), I=1, 16)
1060  FORMAT ('FILE (',I6,') NOT FOUND IN ',16A2)
      GO TO 10  /* 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
120   WRITE(1,1070) WRDNUM,FILNUM,(SEGDIR(I),I=1,16)
1070  FORMAT ('WORD',I6,' NOT IN FILE (',I6,') IN ',16A2)
      GO TO 10  /* RE-TRY
C
      END
```

This program, stored as SEGREAD.FTN, may be compiled, loaded, and run with the following dialog:

```
OK, FTN SEGREAD
0000 ERRORS [<.MAIN.>FTN-REV18.4]
OK, LOAD
[LOAD rev 19.0.1]
$ LO SEGREAD
$ LI
LOAD COMPLETE
$ SA
$ EXEC
TYPE SEGMENT DIRECTORY NAME
SEGDIR
TYPE FILE NUMBER
0
TYPE WORD NUMBER
1
WORD     1 OF FILE (     0) IN segdir                   CONTAINS      0
OK,
```

6
The Pascal Interface

INTRODUCTION

To call a standard subroutine from Pascal, first declare it as an external procedure in the format:

```
PROCEDURE sub-name[([VAR] arg:type[; [VAR] arg:type]...)];EXTERN;
```

Call it with its name and the argument-names used in the program:

```
sub-name[(data-name [,data-name]...)];
```

Note

> In the rest of this guide, subroutine call formats are always given as CALL sub-name [(identifier)...]. From Pascal, however, the word CALL must be omitted.

To declare a function, include the type of value returned by the function:

```
FUNCTION function-name[([VAR] arg: type; [ arg:type]...)]: type;
              EXTERN;
```

Call it with a format such as one of the following:

```
IF function-name(data-name ...)  = X THEN ...;

X = function-name(data-name...);
```

Note

Remember that any arguments that are supplied or changed by the subroutine must be declared as VAR.

DATA TYPES

Table 6-1 summarizes the argument types of FORTRAN and PL1G subroutines and functions that can be called from Pascal. The following is a discussion of these argument types, as well as some generic types, and how they relate to Pascal data types and structures.

INTEGER*2 or FIXED BIN(15)

The INTEGER*2 expected by FORTRAN subroutines is PL1G's FIXED BIN, also called FIXED BIN(15). It must be declared in Pascal programs as INTEGER.

Sample Program 1 illustrates a call to the FORTRAN subroutine SRCH$$, which expects an INTEGER*2 argument. Sample Program 4 calls the PL1G subroutine GV$GET, which needs a FIXED BIN argument.

INTEGER*4 or FIXED BIN(31)

The INTEGER*4 expected by FORTRAN subroutines is PL1G's FIXED BIN(31). Since the INTEGER type in Pascal has a length of only 16 bits, these longer integers must be declared as a subrange. For example, such an operand might be declared as:

```
TYPE INT4 = [-65565 ..  +65565];
```

To define a 32-bit integer, the numbers within brackets must have an absolute value greater than 32768. The absolute value may range as high as 2147483647.

Sample Program 2 calls the FORTRAN subroutine RNUM$A, which expects an INTEGER*4 argument.

Table 6-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 6-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

Integer Arrays

An integer array expected by a FORTRAN subroutine should be declared as an array of numbers or of characters in Pascal, depending on the type of information expected. Sample Program 6 calls the subroutine TIMDTP, which returns an integer array with information of both data types.

Multidimensional arrays should not be passed between FORTRAN and Pascal, because columns and rows will be reversed.

ASCII Character (String or Array)

An ASCII string expected by a FORTRAN subroutine should be declared as a literal or an array of characters in Pascal. Sample Program 3 illustrates passing an ASCII string to the subroutine DELE$A.

LOGICAL

LOGICAL arguments expected by a FORTRAN subroutine should be declared in Pascal as INTEGER. The arguments must have a value of 0 (false) or 1 (true).

Sample Program 3 illustrates a call to the function DELE$A, which returns a logical value. The example for YSNO$A in Chapter 12 also illustrates a call to a logical function from Pascal.

REAL, REAL*4, or FLOAT BIN(23)

This data type should be declared as REAL in Pascal. Constants passed as real arguments to FORTRAN functions should be in scientific format (x.xEyy).

Sample Program 5 passes a REAL argument to the subroutine RND$A.

REAL*8

FORTRAN subroutines that expect arguments of this type may not be called from Pascal.

CHARACTER(n)NONVARYING

This argument type, usually declared simply as CHARACTER(n), may be declared in Pascal as ARRAY [1..n] OF CHAR. A call from Pascal using a CHAR(80)NONVAR argument is given in the example for CL$GET in Chapter 10.

CHARACTER(*)VARYING

This PL1G data type is implemented as a record structure, with the actual number of characters followed by those characters. Thus the structure of a CHAR(*)VAR argument may be represented by the following box:

```
 _______________________________
|0   5  |A   B   C   D   E   |
|___|___|___|___|___|___|___|___|
  COUNT   CHARACTER STRING
```

To declare a comparable structure in Pascal, therefore, requires a record, containing a 16-bit character count plus a character array.

Because of the argument format expected by Pl1G, CHAR(*)VAR arguments may never be passed as literals.

As an example, if the character string to be passed to PL1G is 28 characters long, then the Pascal operand might be constructed this way:

```
    RECORD
        BCOUNT: INTEGER;
        VARNAME: ARRAY[1..28] OF CHAR;
    END;
```

Sample Program 4 calls the Pl1G subroutine GV$GET, which expects two CHAR(*)VAR arguments.

POINTER

A POINTER type expected by a PL1G subroutine may be declared as a pointer in Pascal also. Sample Program 7 calls a subroutine that expects a pointer.

BIT(1)

PL1G subroutines that use this argument type may not be called from Pascal programs, unless the argument is BIT(1)ALIGNED. Then the argument may be passed as a Boolean operand. PL1G's '0'B may then be read as 0 in Pascal, and PL1G's '1'B as -1.

USING SYSCOM TABLES

Subroutine descriptions in this guide sometimes refer to codes with names in the format x$yyyy, where x and y are letters. There are three groups of these codes.

Error codes have names in the format E$yyyy. These equivalents should be inserted in the Pascal program with the statement:

```
%INCLUDE 'SYSCOM>ERRD.INS.PASCAL';
```

This statement should be inserted into the CONST declaration. The equivalents for these error codes are in Appendix D and in the file SYSCOM>ERRD.INS.PASCAL.

Key codes have names in the format K$yyyy. These equivalents should be inserted in the program with the statement:

```
%INCLUDE 'SYSCOM>KEYS.INS.PASCAL';
```

This statement should also be inserted into the CONST declaration. The equivalents for these key codes are in Appendix C and in the file SYSCOM>KEYS.INS.PASCAL.

Some subroutines in VAPPLB use argument codes in the form A$yyyy. These equivalents should be inserted in the Pascal program with the statement:

```
%INCLUDE 'SYSCOM>A$KEYS.INS.PASCAL';
```

following the CONST declaration. The numeric equivalents of these codes are listed in the table at the end of Chapter 12 and in the file SYSCOM>A$KEYS.INS.PASCAL.

Sample Program 1 illustrates the use of key codes.

SAMPLE PROGRAMS

Program 1 — Using INTEGER*2 and SYSCOM Keys

```
PROGRAM SRCH_CAL;
{                                                }
{THIS PROGRAM CALLS THE SUBROUTINE SRCH$$ TO CHECK }
{ON THE EXISTENCE OF A FILE.                       }
{                                                }
CONST
     %INCLUDE 'SYSCOM>KEYS.INS.PASCAL';
     %INCLUDE 'SYSCOM>ERRD.INS.PASCAL';
TYPE STRING = ARRAY[1..6] OF CHAR;
```

```
VAR CODE: INTEGER;
PROCEDURE SRCH$$(A:INTEGER; B:STRING; C:INTEGER; D:INTEGER;
              E:INTEGER; VAR F:INTEGER);EXTERN;
BEGIN
    SRCH$$ (K$EXST+K$IUFD, 'CTRLFL', 6, 0, 0, CODE);
    WRITELN ('SEARCH CODE IS: ', CODE);
    END.
```

This program may be compiled, loaded, and run with the following dialog. If the file CTRLFL is not found, the resulting return code will be 15, as shown below.

```
OK, PASCAL SRCH
0000 ERRORS (PASCAL-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO SRCH
$ LI PASLIB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

SEARCH CODE IS:         15
OK,
```

Program 2 — Using an INTEGER*4 Argument

```
OK, SLIST INT4.PASCAL

PROGRAM INT4;
{                                                                    }
{THIS PROGRAM CALLS THE SUBROUTINE RNUM$A TO VERIFY AN INTEGER*4.    }
{                                                                    }
CONST
%INCLUDE 'SYSCOM>A$KEYS.INS.PASCAL';
TYPE STRING= ARRAY[1..14] OF CHAR;
TYPE INT4 = -100000 .. +100000;
VAR MSG: STRING;
    CODEVALUE: INT4;
PROCEDURE RNUM$A(M:STRING;L:INTEGER;N:INTEGER; VAR V:INT4);EXTERN;
BEGIN
    MSG := 'ENTER A NUMBER';
    RNUM$A (MSG, 14, A$DEC, CODEVALUE);
    WRITELN('NUMBER IS: ', CODEVALUE);
    END.
```

This program, compiled and stored as INT4.PASCAL, may be loaded and run with the following dialog:

```
OK, SEG -LOAD
[SEG rev 19.0]
$ LO INT4
$ LI PASLIB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC


ENTER A NUMBER: Q
  Illegal number (RNUM$A)
ENTER A NUMBER: 11223344556677889900
  Too many digits (RNUM$A)
ENTER A NUMBER: 123456789
NUMBER IS:  123456789
OK,
```

Program 3 -- Calling a Logical Function

```
PROGRAM SUBCALL;
{                                                                  }
{THIS PROGRAM CALLS THE LOGICAL FUNCTION DELE$A TO DELETE A FILE.  }
{                                                                  }
TYPE STRING= ARRAY[1..6] OF CHAR;
VAR FILENAME: STRING;
    THE_COUNT:INTEGER;
{                                                                  }
{THE NEXT FUNCTION WILL RETURN A                                   }
{VALUE OF EITHER 1 (DELETE SUCCESSFUL)                             }
{OR 0 (UNSUCCESSFUL)                                               }
{                                                                  }
FUNCTION DELE$A( A:STRING; K:INTEGER):INTEGER;EXTERN;
BEGIN
    FILENAME := 'CTRLFL';
    THE_COUNT := 6;
    IF DELE$A (FILENAME, THE_COUNT) = 1 THEN
           WRITELN('FILE DELETED')
    ELSE WRITELN('NO GO');
    WRITELN('END OF RUN')
    END.
```

This program, stored as LOGICAL.PASCAL, may be compiled, loaded, and run with the following dialog. If the file CTRLFL exists, the first message will be displayed; otherwise the second message will appear.

```
OK, PASCAL LOGICAL

0000 ERRORS (PASCAL-REV19.0)
OK, SEG -LOAD
[SEG rev 19.0]
$ LO LOGICAL
$ LI PASLIB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC
FILE DELETED
END OF RUN
OK, SEG LOGICAL
Not found. CTRLFL (DELE$A)
NO GO
END OF RUN
OK,
```

Program 4 -- Using CHAR(*)VAR Arguments

The following program returns the value of a global variable set with DEFINE_GVAR. For more information, see the CPL User's Guide or the chapter on CPL files in the Prime User's Guide.

```
OK, SLIST CHARVAR.PASCAL

PROGRAM CHRVR;
TYPE CHARVAR = RECORD
               NCHARS: INTEGER;
               STRING1:  ARRAY[1..4] OF CHAR
               END;
VAR VARSIZE, CODE, K: INTEGER;
    VARVALUE, VARNAME: CHARVAR;
PROCEDURE GV$GET(A:CHARVAR; VAR B:CHARVAR; C:INTEGER; D:INTEGER);
                  EXTERN;
BEGIN;
    VARNAME.NCHARS := 4;
    VARNAME.STRING1 := '.MAX';
    VARSIZE := 4;
    GV$GET (VARNAME, VARVALUE, VARSIZE, CODE);
    K := 1;
    WRITE('SIZE OF MAX IS  ');
```

```
 FOR K := 1 TO VARVAL.NCHARS DO
   WRITE(VARVALUE.STRING1[K]);
 WRITELN;
 WRITELN('ERROR CODE IS ',CODE);
END.
```

To compile and load this program, stored as CHARVAR.PASCAL, use the following dialog:

```
OK, PASCAL CHARVAR
0000 ERRORS (PASCAL-REV19.0)
OK, SEG -LOAD
[SEG rev 19.0]
$ LO CHARVAR
$ LI PASLIB
$ LI
LOAD COMPLETE
$ Q
```

Before this program is run, a global variable file containing the variable .MAX must be defined:

```
OK, DEFINE_GVAR ANNE>GVARFILE
OK, SEG CHARVAR


SIZE OF MAX IS  100
ERROR CODE IS          0
OK,
```

Program 5 — Using a REAL*4 Argument

```
OK, SLIST RANDOM.PASCAL
```

```
PROGRAM RANDOM;
{                                                      }
{ THIS PROGRAM GENERATES TEN RANDOM NUMBERS, STARTING  }
{ FROM A SEED INCLUDED IN THE PROGRAM                  }
{                                                      }
VAR SEED1, THISONE: REAL;
    K: INTEGER;
FUNCTION RAND$A(VAR SEED: REAL): REAL; EXTERN;
BEGIN
   SEED1 := 1.2E-1;
   K := 0;
   FOR K := 1 to 10 DO
       BEGIN
```

```
           THISONE := RAND$A(SEED1);
           WRITELN(K, ':', THISONE);
           END
    END.
```

This program, compiled and stored as RANDOM.BIN, may be loaded and run with the following dialog:

```
    OK, SEG -LOAD
    [SEG rev 19.0]
    $ LO RANDOM
    $ LI PASLIB
    $ LI VAPPLB
    $ LI
    LOAD COMPLETE
    $ EXEC


             0:          7.216268E-01
             1:          3.840753E-01
             2:          1.552343E-01
             3:          2.418942E-02
             4:          5.516532E-01
             5:          6.372356E-01
             6:          1.963481E-02
             7:          2.397342E-03
             8:          2.921368E-01
             9:          9.439590E-01
    OK,
```

Program 6 -- Using an Integer Array

This program calls the subroutine TIMDAT to retrieve system and user information. Since the array CHARARRAY will return both character and numeric data, it is defined twice by means of the CASE statement.

```
OK, SLIST TIMDTP.PASCAL

PROGRAM TIMDTP;

TYPE CHARARRAY = ARRAY[1..30] OF CHAR;
CASEVALUE = (A1,A2);
(*                                                          *)
    TABLE = RECORD CASE I : CASEVALUE OF
       A1 : (J1 : CHARARRAY);
       A2 : (J2 : RECORD MMDDYY: ARRAY[1..6] OF CHAR;
                  TIME_MIN : INTEGER;
                  TIME_SEC : INTEGER;
                  TIME_TCK : INTEGER;
                  CPU_SEC : INTEGER;
                  CPU_TCK: INTEGER;
                  DISK_SEC : INTEGER;
                  DISK_TCK : INTEGER;
                  TCK_SEC : INTEGER;
                  USER_NUM : INTEGER;
                  USERNAME : ARRAY [1..32] OF CHAR;
       END;)
    END;
(*                                                        *)
VAR TABLE1 : TABLE;
       I : CASEVALUE;
PROCEDURE TIMDAT(VAR A:CHARARRAY; B:INTEGER);EXTERN;
(*                                                        *)
BEGIN
    I := A1;                    (*CHARACTER ARRAY*)
    TIMDTP(TABLE1.J1,28);
    I := A2;                    (*RECORD, CHAR and INTEGER*)
    WITH TABLE1.J2 DO
      BEGIN
    WRITELN('DATE IS               ', MMDDYY);
    WRITELN('SECONDS ELAPSED   ',TIME_SEC);
    WRITELN('TICKS ELAPSED     ',TIME_TCK);
    WRITELN('CPU SECONDS USED  ', CPU_SEC);
    WRITELN('CPU TICKS         ', CPU_TCK);
    WRITELN('DISK SECONDS USED ', DISK_SEC);
    WRITELN('USER NAME              ', USERNAME);
    END
END.
```

To compile, load, and run this program, stored as TIMDTP.PASCAL, use the following dialog:

```
OK, PASCAL TIMDTP
0000 ERRORS (PASCAL-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO TIMDTP
$ LI PASLIB
$ LI
LOAD COMPLETE
$ EXEC

DATE IS                012082
SECONDS ELAPSED            15
TICKS ELAPSED             102
CPU SECONDS USED           44
CPU TICKS                 223
DISK SECONDS USED          57
USER NAME                ANNE
OK,
```

Program 7 -- Using a Pointer Argument

```
OK, SLIST PTR.PASCAL

PROGRAM ACLCTL;
{                                             }
{ THIS PROGRAM CREATES AN ACL FOR THE FILE    }
{ RISKFILE, OR, IF AN ACL ALREADY EXISTS,     }
{ RETURNS AN ERROR MESSAGE.                   }
{                                             }
TYPE STRING = ARRAY[1..7] OF CHAR;
TYPE CHARVAR = RECORD
    NUMBER: INTEGER;
    FILENAME: STRING;
    END;
TYPE ACL = RECORD
             VERSION: INTEGER;
             ENTRY_COUNT:INTEGER;
             ENTRIES: ARRAY[1 .. 2] OF CHARVAR;
TYPE ACL_PTR = ^ACL;
```

```
VAR KEY: INTEGER;
    NAME: CHARVAR;
    CODE:INTEGER;
    THISPTR : ACL_PTR;
    RISKFILE: ACL;
PROCEDURE AC$SET(A:INTEGER;B:CHARVAR;C*ACL_PTR; D:INTEGER);
                 EXTERN;
{                                                         }
BEGIN
    KEY := 7;       {7 = K$CREA}
    NAME.NUMBER := 7;
    NAME.FILENAME := 'ACLTEST';
    RISKFILE.VERSIN := 2;
    RISKFILE.ENTRY_COUNT:= 1;
    RISKFILE.ENTRIES[1].NUMBER := 7;
    RISKFILE.ENTRIES[1].FILENAME := 'RSKFILE';
    NEW(THISPTR);
    THISPTR^ := RISKFILE;
    AC$SET (KEY, NAME, THISPTR, CODE);
    WRITELN ('CODE IS: ', CODE)
END.
```

This program, stored as PTR.PASCAL, may be compiled, loaded, and executed with the following dialog:

```
OK, PASCAL PTR
0000 ERRORS (PASCAL-REV19.0)
OK, SEG -LOAD
[SEG rev 19.0.1]
$ LO PTR
$ LI PASLIB
$ LI
LOAD COMPLETE
$ EXEC

CODE IS:          0
OK,
```

7
The PL/I Subset G Interface

INTRODUCTION

To call an external subroutine from PL/I subset G (PL1G), first declare the subroutine as an external procedure in the format:

```
    DECLARE sub-name EXTERNAL ENTRY[(type [,type]...)];
```

where sub-name is the subroutine name without quotes, and type is the type of the argument expected.

To call the subroutine, use the format:

```
    CALL sub-name[(identifier [,identifier]...)];
```

where identifier may be a constant or a data name.

To declare a function, use this format:

```
    DECLARE function-name EXTERNAL ENTRY[(type ...)] RETURNS (type);
```

Call it as an expression in a format like one of these:

```
    IF logical-function[(identifier...)] = 0 THEN ...;

    IF function-name[(identifier...)] = X THEN ...;
```

THE OPTIONS (SHORTCALL) DECLARATION

The OPTIONS(SHORTCALL) declaration is useful for calling PMA procedures with the PMA instruction JSXB instead of the more common PCL instruction. A procedure call of this type is faster than one using PCL. However, the called procedure must be written to expect this kind of call. In Rev. 18 and Rev. 19, the only system subroutine that can (and must) be declared in this way is MKONU$.

The format of this declaration is:

```
DECLARE procedure-name ENTRY OPTIONS(SHORTCALL [stack-size] );
```

stack-size specifies the extra space needed for the calling procedure's stack. The default size is 8.

The call does not generate a new stack for storage, as does PCL. The calling procedure's stack space is used. Thus it may be necessary to specify stack size in the declaration in order to enlarge the calling stack. For example, MKONU$ requires an 28-word stack, so the user's stack must be large enough to accomodate this requirement. If stack size is not large enough, the return from the subroutine will cause unpredictable error messages.

Arguments may be used with the SHORTCALL option. The computer will set up the L register to point to a vector containing the addresses of the arguments, or, in the case of one argument, to the address of the argument itself. No type checking is done. For Rev. 19, there are no standard subroutine calls that require both SHORTCALL and argument passing.

DATA TYPES

Table 7-1 summarizes the argument types of FORTRAN subroutines and functions that can be called from PL1G. The following is a discussion of these argument types, as well as some generic types, and how they relate to PL1G data types and structures. The PL1G CHAR(*)VARYING argument type is discussed briefly.

INTEGER*2

The INTEGER*2 expected by FORTRAN subroutines is PL1G's FIXED BIN, also called FIXED BIN(15). Sample Program 1 includes a call to the subroutine SRCH$$, which expects an INTEGER*2 argument.

Table 7-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 7-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

INTEGER*4

The INTEGER*4 expected by FORTRAN subroutines is PL1G's FIXED BIN(31). Sample Program 2 calls the FORTRAN subroutine RNUM$A, which expects an INTEGER*4 argument.

REAL or REAL*4

This FORTRAN data type should be declared as FLOAT BIN, also called FLOAT BIN(23) in PL1G. Constants passed to a FORTRAN function that expects REAL arguments should be in scientific format (x.xE±yy).

Sample Program 3 calls RAND$A, which expects a real number.

REAL*8

The REAL*8 argument expected by a FORTRAN subroutine should be declared in PL1G as FLOAT BIN(47). It should be in scientific format (x.xE±yy).

Integer Arrays

An integer array expected by a FORTRAN subroutine should be declared, according to the kind of information passed, either as a FIXED BINARY(15) array or as a character array in PL1G:

```
    DECLARE X(1:n) FIXED BIN(15);

    DECLARE X(1:n) CHAR;

    DECLARE X CHAR(n);
```

Multidimensional arrays cannot be passed between FORTRAN and PL1G.

ASCII Character (String or Array)

An ASCII string expected by a FORTRAN subroutine should be declared in PL1G as a literal or as CHAR(n)NONVARYING.

LOGICAL

LOGICAL arguments expected by a FORTRAN subroutine should be declared in PL1G as FIXED BIN(15). The arguments must have a value of 0 (false) or 1 (true). Note that FORTRAN logical functions cannot be called as functions in PL1G for this reason, and must be called as subroutines.

Sample Program 4 calls the function DELE$A, which returns a logical value.

CHARACTER(*)VARYING

This argument is expected only by PL1G subroutines. It should be declared as CHAR(n)VARYING, and passed only as a data name, not as a literal. No other special steps are needed to pass CHAR(*)VARYING from a PL1G program.

Sample Program 5 calls the Pl1G subroutine GV$GET, which expects a CHAR(*)VARYING argument.

CHARACTER(n)NONVARYING, POINTER, BIT(1)

These arguments are expected only by PL1G standard subroutines. They should be declared the same way in the calling routine.

USING SYSCOM TABLES

Subroutine descriptions in this guide sometimes refer to codes with names in the format x$yyyy, where x and y are letters. The code names may be used in the program instead of numeric values. There are three groups of these codes.

Error codes have names in the format E$yyyy. These equivalents should be inserted in the PL1G program before the subroutine declaration with the statement:

```
    %INCLUDE 'SYSCOM>ERRD.INS.PL1';
```

The equivalents for these key codes are in Appendix D and in the file SYSCOM>ERRD.INS.PL1.

Key codes have names in the format K$yyyy. These equivalents should be inserted in the program with the statement:

```
    %INCLUDE 'SYSCOM>KEYS.INS.PL1';
```

The equivalents for these key codes are in Appendix C and in the file SYSCOM>KEYS.INS.PL1.

Some subroutines in VAPPLB use argument codes in the form A$yyyy. These codes should also be inserted with the statement %INCLUDE 'SYSCOM>A$KEYS.INS.PL1'. They are listed in the table at the end of Chapter 12 on VAPPLB, or in the file SYSCOM>A$KEYS.INS.PL1.

Sample Program 1 illustrates the use of key codes.

SAMPLE PROGRAMS

Program 1 — Using INTEGER*2 and SYSCOM Keys

```
SUBS: PROCEDURE OPTIONS(MAIN);
  /*                                                           */
  /*  A PROGRAM TO CALL THE SUBROUTINE SRCH$$ TO VERIFY THE    */
  /*  EXISTENCE OF FILE CTRLFL
  /*
  /*                                                           */
  %INCLUDE 'SYSCOM>KEYS.INS.PL1';
  %INCLUDE 'SYSCOM>ERRD.INS.PL1';
  DCL CODE FIXED BIN;
  DCL SRCH$$ EXTERNAL ENTRY (FIXED BIN, CHAR(6), FIXED BIN,
                             FIXED BIN, FIXED BIN, FIXED BIN);
  /*                                                           */
  CALL SRCH$$(K$EXST+K$IUFD, 'CTRLFL', 6, 0, 0, CODE);
  PUT SKIP LIST ('CODE IS: ', CODE);
  END SUBS;
```

This program, stored as SRCH.PL1G, may be compiled, loaded, and run with the following dialog. If the file CTRLFL does not exist, the code 15 will be returned.

```
OK, PL1G SRCH
0000 ERRORS (PL1G-REV19.0)

OK, SEG -LOAD
[SEG Rev 19.0]
$ LO SRCH
$ LI PL1GLB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

CODE IS:              15
OK,
```

Program 2 — Using INTEGER*4

```
RNUM: PROCEDURE OPTIONS(MAIN);
  /*                                              */
  /*A PROCEDURE TO CALL SUBROUTINE RNUM$A TO      */
  /*VERIFY A LONG INTEGER                         */
  /*                                              */
```

```
        %INCLUDE 'SYSCOM>A$KEYS.PL1';
        DCL CODE FIXED BIN(31);
        DCL RNUM$A EXTERNAL ENTRY (CHAR(14), FIXED BIN, FIXED BIN,
                                  FIXED BIN(31));
        CALL RNUM$A ('ENTER A NUMBER', 14, A$DEC, CODE);
        PUT SKIP LIST ('NUMBER IS', CODE);
        END RNUM;
```

This program, stored as INT4.PL1G, may be compiled, loaded, and run with the following dialog:

```
    OK, PL1G INT4
    0000 ERRORS (PL1G-REV19.0)

    OK, SEG -LOAD
    [SEG rev 19.0]
    $ LO INT4
    $ LI PL1GLB
    $ LI VAPPLB
    $ LI
    LOAD COMPLETE
    $ EXEC

    ENTER A NUMBER: Q
      Illegal number (RNUM$A)
    ENTER A NUMBER: 123456789123456789
      Too many digits (RNUM$A)
    ENTER A NUMBER: 12345

    NUMBER IS              12345
    OK,
```

Program 3 — Using REAL*4

```
    OK, SLIST RANDOM.PL1G
    RANDOM : PROCEDURE OPTIONS(MAIN);
      /*
      /* A PROGRAM TO CALL RAND$A TO GENERATE RANDOM NUMBERS
      /*                                                       */
      DCL K FIXED BIN;
      DCL SEED STATIC FIXED BIN(31) INITIAL (1);
      DCL REAL4 FLOAT;
      DCL RAND$A EXTERNAL ENTRY (FIXED BIN(31)) RETURNS (FLOAT);
      /*                                                       */
      DO K = 1 TO 10;
      REAL4 = RAND$A(SEED);
      PUT SKIP LIST(REAL4);
      END;
    END RANDOM;
```

This program may be compiled, loaded, and run with the following dialog:

```
OK, PL1G RANDOM
0000 ERRORS (PL1G-REV19.0)
OK, SEG -LOAD
[SEG rev 19.0]
$ LO RANDOM
$ LI PL1GLB
$ LI VAPPLB
$ LI
LOAD COMPLETE

$ EXEC

 7.826369E-06
 1.315377E-01
 7.556052E-01
 4.586501E-01
 5.327672E-01
 2.189591E-01
 4.704461E-02
 6.788645E-01
 6.792963E-01
 9.346929E-01
OK,
```

Program 4 — Calling a Logical Function

```
LOGI: PROCEDURE OPTIONS(MAIN);
  /*                                                  */
  /*A PROCEDURE TO CALL FUNCTION DELE$A TO            */
  /*DELETE A FILE AND VERIFY THAT IT DID              */
  /*                                                  */
  DCL DELE$A EXTERNAL ENTRY(CHAR(6),FIXED BIN) RETURNS(FIXED BIN);
  IF DELE$A ('CTRLFL', 6) = 1 THEN
       PUT SKIP LIST ('FILE DELETED');
  ELSE PUT SKIP LIST ('NO GO');
  END LOGI;
```

This program, stored as LOGICAL.PL1G, may be compiled, loaded, and run with the following dialog if CTRLFL does not exist.

```
OK, PL1G LOGICAL
0000 ERRORS (PL1G-REV19.0)

OK, SEG -LOAD
[SEG REV19.0]
$ LO LOGICAL
$ LI PL1GLB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

Not found. CTRLFL (DELE$A)

NO GO
OK,
```

Program 5 — Using CHAR(*)VARYING Arguments

```
OK, LIST GVAR.PL1G

GVAR:   PROCEDURE OPTIONS(MAIN);
  /*
  /* A PROGRAM TO ASCERTAIN THE VALUE OF A GLOBAL VARIABLE
  /*                                                            */
  DCL VAR_NAME STATIC CHAR(4)VAR INIT('.MAX');
  DCL VAR_VALUE CHAR(4)VAR;
  DCL VALUE_SIZE STATIC FIXED BIN INITIAL(4);
  DCL CODE FIXED BIN;
  DCL GV$GET EXTERNAL ENTRY (CHAR(*)VAR, CHAR(*)VAR,
                             FIXED BIN, FIXED BIN);
  /*                                                            */
  CALL GV$GET(VAR_NAME, VAR_VALUE, VALUE_SIZE, CODE);
  PUT SKIP LIST ('MAX IS', VAR_VALUE);
  PUT SKIP LIST ('CODE IS: ', CODE);
END GVAR;
```

This program, compiled and stored as GVAR.PL1, may be loaded and run with the following dialog, providing that a global variable file has been defined as explained in The CPL User's Guide.

```
OK, SEG -LOAD
[SEG REV19.0]
$ LO GVAR
$ LI PL1GLB
$ LI
LOAD COMPLETE
$ EXEC

MAX IS 100
CODE IS:              0
OK,
```

8 The PMA Interface

INTRODUCTION

Table 8-1 summarizes the argument types of FORTRAN and PL1G subroutines that can be called from PMA. PRIMOS subroutines are particularly useful to the PMA programmer for doing device I/O, for displaying data on the terminal, and for doing file manipulation.

To call a subroutine, simply write:

```
    CALL sub-name
```

Then, on succeeding lines, list the arguments to be passed, preceded by AP for V-mode or DAC for R-mode and followed in V-mode by S or SL as discussed below.

External functions should not be called from PMA. However, most functions in this guide may also be called as subroutines.

Calling Subroutines from V-mode and I-mode PMA

When PMA calls an external subroutine in V-mode or I-mode, arguments are passed by reference using the AP instruction. Each AP instruction uses S as its second operand, except the last, which uses SL. Examples of V-mode calls are given in the first set of sample programs below. The same programs may be used in I-mode with SEGR instead of SEG at the beginning.

Table 8-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G
1 bit	-*-	-*-	-*-	-*-	-*-	(1) Bit Bit(1)
16-bit Half-word	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin(23)
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n)
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	Char(n) Varying
(7) 48-bits 3 Half-words	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer

* Not available.

Notes to Table 8-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

Calling Subroutines from R-mode PMA

When PMA calls an external subroutine in R-mode, arguments are passed by reference using the DAC instruction. If there is more than one argument, the last DAC instruction is followed by DATA 0. (This is a convention of the operating system, and not an architectural feature.) If there is only one argument, DATA 0 must not be used. Examples of R-mode calls are given in the second set of sample programs below.

DATA TYPES

Refer to the Assembly Language Programmer's Guide for more details on the following data types.

INTEGER*2 or FIXED BIN(15)

FORTRAN's INTEGER*2 is PL1G's FIXED BIN(15), also called just FIXED BIN. This 16-bit argument is the one-word (single-precision) data type that is defined by default with the BSS, DYNM, BSZ, OCT, or DATA statement in PMA.

Sample Programs 2 and 7 use INTEGER*2 arguments.

INTEGER*4 or FIXED BIN(31)

This 32-bit argument expected by FORTRAN or PL1G is the double-word, double-precision data type that is defined with BSS 2, DYNM x (2), or DATA xxxxL. Sample Programs 3 and 8 use this data type.

REAL*4 or REAL

This 32-bit argument expected by FORTRAN is the single-precision floating-point data type that is defined by any DATA item with a decimal point or scientific notation (nnEnn), BSS 2, or DYNM x (2).

REAL*8

This 64-bit argument expected by FORTRAN is the double-precision floating-point data type that is defined by any DATA item with a decimal point or scientific notation and with D appended to it, by BSS 4, or by DYNM x (4).

Integer Array

This may be passed as any data type.

LOGICAL

This FORTRAN data type is a 16-bit integer, with a value of 1 for true or 0 for false. Sample Program 4 uses a LOGICAL data type.

ASCII Characters (String)

ASCII characters can be passed as a constant string enclosed in apostrophes after the DATA statement plus the letter C, for example, DATA C'STEP 1'. It may also be enclosed in any delimiter after the BCI statement. The maximum number of characters after C is 32. The maximum after BCI is the number that will fit on the same statement line.

CHARACTER(*)VARYING

This PL1G data type is implemented as a record structure, with the actual number of characters followed by those characters. The elements may be pictured as follows:

```
 _______________________________
|0   5  |A   B   C   D   E   |
|___|___|___|___|___|___|___|___|
 COUNT    CHARACTER STRING
```

Sample Program 5 uses a CHAR(*)VAR data type.

CHARACTER(n)NONVARYING

This PL1G data type, usually declared simply as CHARACTER(n), consists of n characters. It may be coded in PMA as DATA C'xxx...', or may be passed as a literal. Either item should be n characters long.

BIT(1)

PL1G programs that expect arguments of this type should not be called from PMA unless the argument is declard in PL1G as BIT(1) ALIGNED. In this case it may be treated as a 16-bit integer, with a value of -1 for false.

USING SYSCOM TABLES

Many subroutine descriptions in this guide use, instead of numeric codes, key names in the form x$yyyy where x and y are letters. There are three files in the SYSCOM UFD that are of use in handling these names.

SYSCOM>KEYS.INS.PMA and SYSCOM>ERRD.INS.PMA contain the equivalents of keys and error codes. They should be used instead of numeric values for codes. These keys are explained in Chapter 2. To use these key names in a PMA program, use $INCLUDE SYSCOM>xxxx or $INSERT SYSCOM>xxxx anywhere in a program.

There is no A$KEYS file for PMA, so the numeric values of the codes must be used instead. These codes are in Chapter 12 of this guide, or may be read from the SYSCOM>A$KEYS.INS.FTN file.

Sample Programs 1, 6, and 8 illustrate use of these SYSCOM tables.

DIRECT-ENTRANCE CALLS TO PRIMOS — THE PCL INSTRUCTION

V-mode supports direct-entrance calls to certain procedures. Routines such as SRCH$$, TNOU, or PRWF$$ can be invoked directly by this mechanism. In V-mode, the CALL instruction is really a pseudo-op that contains an EXT (external) declaration and a PCL (procedure call) instruction. The PCL first searches to see whether the called routine is a name in PRIMOS' gate segment. If so, the subroutine code does not have to be loaded into the user's memory space. If the procedure name is not in the gate segment, PCL looks in the libraries loaded by SEG. Direct-entrance calls are available only from V-mode and I-mode programs and will be correctly set up by loading the V-mode FTN library with LI after SEG is invoked.

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

Sample Program 4 illustrates a call using the PCL instruction. There is no advantage to using this method rather than using CALL. The distinction between these calls and normal subroutine calls is presented only for background.

Under R-mode memory images on PRIMOS II or PRIMOS III, all operating system subroutines use the SVC interface described in Appendix H. In R-mode, only experienced programmers should use direct-entry calls in programs, as discussed in the Assembly Language Programmer's Guide.

SAMPLE PROGRAMS IN V-MODE

Program 1 — Using SYSCOM Keys

This program calls SRCH$$ to verify the existence of the file CTRLFL, using the key K$EXST. The program then calls TOVFD$ to print the error code returned by SRCH$$.

```
           SEG                          THIS IS V-MODE
  MAIN     CALL TNOUA                   DISPLAY CHARACTERS:
           AP   =C'CODE ',S               FIRST ARGUMENT
           AP   =5,SL                     SECOND ARGUMENT
  $INSERT SYSCOM>KEYS.INS.PMA
           CALL  SRCH$$                 CALL SEARCH:
           AP     =K$EXST+K$IUFD,S        KEY ARGUMENT
           AP    =C'CTRLFL',S             FILENAME ARG
           AP    =6,S                     LENGTH ARG
           AP    =0,S                     FUNIT ARG
           AP    =0,S                     TYPE ARG
           AP    CODE,SL                  LAST ARG
           CALL TOVFD$                  PRINT INTEGER:
           AP    CODE,SL                  ONLY ARG
           CALL TONL                    NEWLINE
           CALL EXIT                    END GRACEFULLY
           LINK                         DEFINE DATA:
  CODE     BSS    1                       16-BIT INTEGER
  ECB$     ECB  MAIN
           END  ECB$
```

To assemble, load, and run this program, stored as SRCHV.PMA, use the following dialog:

```
OK, PMA SRCHV
0000 ERRORS (PMA-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO SRCHV
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

CODE  0
OK,
```

Program 2 — Using INTEGER*2 Arguments

This program calls E$11 (Appendix G) to do exponentiation, then calls TOVFD$ to print the 16-bit result. The program uses the DYNM data definition to put 16-bit integers on a stack.

```
         SEG                 THIS IS V-MODE
         DYNM ITEM,Y         16-BIT INTEGERS
MAIN     LDA  =5             PUT 5 IN REGISTER A
         STA  ITEM
         LDA  =2
         STA  Y
         LDA  ITEM           LOAD NUMBER TO BE SQUARED
STRT     CALL E$11           CALL SUBROUTINE FOR EXPONENTIATION
         AP   Y,SL              Y IS POWER TO BE USED
         STA  ITEM           STORE RESULT IN ITEM
         CALL TNOUA          CALL SUBROUTINE TO PRINT MESSAGE
         AP   =C'RESULT ',S     FIRST ARG (MESSAGE)
         AP   =7,SL             SECOND ARG (NO. OF CHARS)
         CALL TOVFD$         CALL SUBROUTINE TO PRINT INTEGER
         AP   ITEM,SL           ONLY ARGUMENT
         CALL TONL           CALL SUBROUTINE FOR NEW LINE
         CALL EXIT           END GRACEFULLY
         LINK
ENTCB    ECB  MAIN
         END   ENTCB
```

To assemble, load, and run this program, stored as TNOUVA.PMA, use the following dialog:

```
OK, PMA TNOUV
0000 ERRORS (PMA-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO TNOUV
$ LI
LOAD COMPLETE
$ EXEC

RESULT 25
OK,
```

Program 3 — Using INTEGER*4

This program calls RNUM$A to accept a 32-bit integer.

```
         SEG                 THIS IS V-MODE
STRT     CALL RNUM$A         CALL SUBROUTINE TO ACCEPT NUMBER
         AP   =C'ENTER A NUMBER',S
```

```
        AP    =14,S
        AP    A$BIN,S
        AP    ITEM,SL
        CALL  EXIT
        LINK
ITEM    BSS  2            32-BIT INTEGER
A$BIN   DATA 9            ACCEPT BINARY ONLY
ECB1    ECB  STRT
        END  ECB1
```

To assemble, load, and run this program, stored as INT4V.PMA, use the following dialog. Since the key A$BIN specifies that a binary number must be entered (See Chapter 12.), an entry of anything but 1's or 0's generates an error message from RNUM$A.

```
OK, PMA INT4V
0000 ERRORS (PMA-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO INT4V
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

ENTER A NUMBER: Q
  Illegal number (RNUM$A)
ENTER A NUMBER: 23
  Illegal number (RNUM$A)
ENTER A NUMBER: 0110
OK,
```

Program 4 — Using Logicals

This program calls TEXTO$ to check whether a filename is valid. It also illustrates use of the PCL instruction.

```
OK, SLIST LOGICAL.PMA

        SEG
        EXT   TEXTO$
MAIN    PCL   TEXTO$
        AP    =C'CTRLFL',S
        AP    =6,S
        AP    LEN,S
        AP    OK,SL
        CALL  TOVFD$    CALL SUBROUTINE TO PRINT OK
        AP    OK,SL
        CALL  TONL      CALL SUBROUTINE FOR NEW LINE
```

```
         CALL EXIT
         LINK
LEN      DATA  6          16-BIT INTEGER
OK       BSS   1          16-BIT INTEGER (LOGICAL)
ECB$     ECB   MAIN
         END   ECB$
```

To assemble, load, and run this program, stored as LOGICAL.PMA, use the following dialog. If the file CTRLFL exists and is successfully deleted, the return code will be 0. Otherwise the code will be 1.

```
OK, PMA LOGICAL
0000 ERRORS (PMA-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO LOGICAL
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC

1
OK,
```

Program 5 — Using CHAR(*)VARYING

This program calls GV$GET, which reads a previously defined global file. Before this program will execute correctly, the global variable file must have been defined with DEFINE_GVAR.

GV$GET can only be called from a program running in V-mode.

```
OK, SLIST CHARVAR.PMA

         SEG
MAIN     CALL  GV$GET
         AP    NAME,S    CHAR*VAR ARG
         AP    VAL,S     CHAR*VAR RETURN ARG
         AP    SIZE,S    ONE-WORD ARG
         AP    CODE,SL   ONE-WORD RETURN ARG
         CALL  TNOU      PRINT CHARACTERS:
         AP    =C'CODE IS ',S
         AP    =8,SL
         CALL  TOVFD$    PRINT NUMBER
         AP    CODE,SL
         CALL  TONL      NEWLINE
         CALL  TNOU
         AP    =C'MAX IS ',S
         AP    =7,SL
         CALL  TNOU
         AP    VAL+1,S  ONLY PRINT SECOND PART OF VAL
         AP    VAL,SL
         CALL  TONL
         CALL  EXIT
         LINK
NAME     DATA  4        ONE-WORD INTEGER +
         BCI   '.MAX'       FOUR-CHAR NAME
VAL      DATA  4        ONE-WORD INTEGER(SUPPLIED) +
         BSS   2            FOUR-CHARACTERS RETURNED
SIZE     DATA  4            16-BIT INTEGER
CODE     BSS   1        16-BIT INTEGER
ECB$     ECB   MAIN
         END   ECB$
```

To assemble, load, and run this program, stored as CHARVAR.PMA, use the following dialog. Before the program can be run successfully, a global variable file containing .MAX must have been defined with the command DEFINE_GVARFILE filename, as explained in the CPL User's Guide.

```
OK, PMA CHARVAR
0000 ERRORS (PMA-REV19.0)

OK, SEG -LOAD
[SEG rev 19.0]
$ LO CHARVAR
$ LI
LOAD COMPLETE
$ EXEC

CODE IS
0
MAX IS
100

OK,
```

SAMPLE PROGRAMS IN R-MODE

Program 6 — Using SYSCOM KEYS

This program does the same thing as Sample Program 1 above.

```
         REL                        THIS IS R-MODE
MAIN
         CALL SRCH$$                CALL SUBROUTINE SRCH
         DAC =K$EXST+K$IUFD           KEY ARG
         DAC =C'CTRLFL'               FILENAME ARG
         DAC =6                       LENGTH ARG
         DAC =0                       FUNIT ARG
         DAC =0                    TYPE ARG
         DAC CODE                  CODE ARG
         DATA 0                  END OF ARGS
         CALL TOVFD$             DISPLAY CODE
         DAC  CODE                  ONLY ARGUMENT
         CALL TONL               NEW LINE
         CALL EXIT               END GRACEFULLY
CODE     BSS    1                DEFINE 16-BIT INTEGER
$INCLUDE SYSCOM>KEYS.INS.PMA
         END
```

To assemble, load, and run this program, stored as SRCH.PMA, use the following dialog. If CTRLFL does not exist, an error code of 15 is returned. (See Appendix D.)

```
OK, PMA SRCH
0000 ERRORS (PMA-REV19.0)

OK, LOAD
[LOAD rev 19.0]
$ LO SRCH
$ LI APPLIB
$ LI
LOAD COMPLETE
$ EXEC

15
OK,
```

Program 7 — Using INTEGER*2

This program does the same thing as Sample Program 2 above.

```
         REL               THIS IS R-MODE
MAIN     LDA  ITEM         LOAD NUMBER TO BE SQUARED
STRT     CALL E$11         CALL SUBROUTINE FOR EXPONENTIATION
         DAC  Y               Y IS POWER TO BE USED
         STA ITEM          STORE RESULT IN ITEM
         CALL TNOUA        CALL SUBROUTINE TO PRINT MESSAGE
         DAC  =C'RESULT '     FIRST ARG (MESSAGE)
         DAC  =7              SECOND ARG (NO. OF CHARS)
         DATA  0           NO MORE ARGUMENTS
         CALL TOVFD$       CALL SUBROUTINE TO PRINT INTEGER
         DAC  ITEM            ONLY ARGUMENT
         CALL TONL         CALL SUBROUTINE FOR NEW LINE
         CALL  EXIT
ITEM     DATA 5            16-BIT INTEGERS
Y        DATA 2
         END
```

To assemble, load, and run this program, stored as TNOUR.PMA, use the following dialog:

```
    OK, PMA TNOUR
    0000 ERRORS (PMA-REV19.0)

    OK, LOAD
    [LOAD rev 19.0]
    $ LO TNOUR
    $ LI
    LOAD COMPLETE
    $ EXEC

    RESULT 25
    OK,
```

Program 8 — Using INTEGER*4 and A$KEYS

This program uses the values in A$KEYS to call RNUM$A, which accepts a 32-bit integer and checks that the integer is in the right format. In this case, the key value is set to 9 for binary input, so the number entered by the user may consist only of 1's and 0's.

```
    OK, SLIST INT4R.PMA

             REL                R-MODE
    STRT     CALL RNUM$A        CALL SUBROUTINE TO ACCEPT NUMBER
             DAC  =C'ENTER A NUMBER'
             DAC  =14           MESSAGE LENGTH
             DAC  A$BIN         SYSCOM>A$KEY FOR BINARY
             DAC  ITEM
             DATA 0             END OF ARGUMENTS
             CALL TONL          CALL SUBROUTINE FOR NEWLINE
             CALL EXIT
    ITEM     BSS  2             32-BIT INTEGER
    A$BIN    DATA 9             16-BIT INTEGER
             END
```

To load this R-mode program, compiled and stored as INT4R.BIN, use the following steps:

```
    OK, LOAD
    [LOAD rev 19.0]
    $ LO INT4R
    $ LI APPLIB
    $ LI
    LOAD COMPLETE
    $ SA
    $ EXEC
```

When this program is run, RNUM$A produces messages similar to the following:

```
ENTER A NUMBER: Q
  Illegal number (RNUM$A)
ENTER A NUMBER: 1122334455
  Illegal number (RNUM$A)
ENTER A NUMBER: 1110000000000001

OK,
```

PART III

PRIMOS Subroutines

9 File Management Subroutines

DEFINITIONS

This section describes some concepts and argument names that are used in Chapter 9. More discussion on file management is provided with SRCH$$ below. Refer to Appendix I for a discussion of file organization prior to Rev. 19.

The subroutines discussed in this chapter are listed on the following page.

Keys

Many subroutines require a key argument, which is numeric. However, all keys to be input by the programmer are specified in this guide in symbolic, rather than numeric, form. These symbolic names are defined in files in the UFD named SYSCOM on the master disk. The key definition files are named KEYS.INS.language. The exact name of the relevant file, if one exists, and how to insert it in a program, is explained for each language in Chapters 3 through 8. The keys are also listed in Appendix C. The programmer is urged to use these symbolic names where possible.

Adding Keys: In call formats, keys may be added, as in this example:

```
CALL SRCH$$ (action + ref + newfil, filnam...)
```

Table 9-1
File Management Functions

Open Files
SRCH$$
TSRC$$

Close Files
SRCH$$

Delete Files
SRCH$$

Search for File
SRCH$$
19 SRSFX$

Manage File Attributes
SATR$$

Find Open Filename
GPATH$

Compare Filenames
NAMEQ$

Change Filename
CNAM$$

Manage Suffixes
APSFX$
SRSFX$

Read/Write
FORCEW
PRWF$$
RDLIN$
WTLIN$

Manage Passwords
GPAS$$
SPAS$$

Manage Segment Directories
SGDR$$

Manage Command Files
COMI$$
COMO$$

Manage R-mode Runfiles
REST$$
RESU$$
SAVE$$

Manage UFDs
Q$READ
Q$SET
ATCH$$
CREA$$
RDEN$$
UPDATE (PRIMOS II)

Since the key names represent numeric values, they may be used as arithmetic expressions, as in this Pascal call:

```
SRCH$$ (K$READ + K$CACC)
```

Keys may be omitted from these expressions unless they are required. The keys may be used in the expression in any order. They are always INTEGER*2.

Error Code or Return Code

The integer return code is a symbolic name for the code returned by a subroutine. It is usually referred to as the error code, but if no errors are encountered the code is returned as 0. The symbolic names are defined in files in the SYSCOM UFD, named ERRD.INS.language. The exact name of the relevant file, if one exists, and how to insert it in a program, is explained for each language in Chapters 3 through 8. Definitions are also given in Appendix D. Error codes are always INTEGER*2.

File System Object

A file system object may be a file, UFD or sub-UFD, a segment directory, or an access category.

Filenames, Pathnames, MFDs, and UFDs

Filenames may be 1 through 32 characters in length, the first character of which must be nonnumeric. Filenames may be composed only of the following characters: A through Z, 0 through 9, _ # $ & * - . /. Names should not begin with a dash (-) or underscore (_). Filenames may not contain embedded blanks.

A UFD (User File Directory) is a directory or subdirectory of files.

A pathname is the name of a file, preceded by as many of its superior UFD-names as is necessary to identify the location of the file. It may be up to 128 characters long. In a pathname, names of all groups except the lowest are followed by a symbol >. If the pathname begins with the MFD (Master File Directory or partition name), this name starts with the symbol <. A complete pathname might be:

```
<TPUBS>ANNE>SOURCE>GVAR.COBOL
```

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

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 <u>subdirectory</u> specifier has the following format:

```
ufdname>subname   [password]
```

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 128 characters, it may cause an error message when passed to a subroutine. Trailing blanks are not allowed in names that are passed as CHAR(*)VARYING strings.

Pathnames 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 pathname in single quotes.

<u>Examples</u>: The following expressions illustrate pathnames, including the required passwords.

ABC File named ABC in home directory.

XYZ>ABC File named ABC in UFD named XYZ.

<INV>XYZ>ABC File named ABC in UFD named XYZ on partition named INV.

<*>XYZ>ABC File named ABC in UFD named XYZ on home partition or MFD.

<5>XYZ>ABC File named ABC in UFD named XYZ on logical disk 5.

*>XYZ>ABC	File named ABC in sub-UFD named XYZ in home directory.
*>XYZ>IJK>ABC	File named ABC in sub-UFD IJK in sub-UFD named XYZ in home directory.
XYZ DEF>ABC	File named ABC in UFD named XYZ with password DEF.
XYZ>ABC	File named ABC in UFD named XYZ.
<INV>XYZ>ABC	File named ABC in UFD named XYZ on volume named INV.
<*>XYZ>ABC	File named ABC in UFD named XYZ on home volume.
<5>XYZ>ABC	File named ABC in UFD named XYZ on logical disk 5.
*>XYZ>ABC	File named ABC in sub-UFD named XYZ in home directory.
*>XYZ>IJK>ABC	File named ABC in sub-UFD IJK in sub-UFD named XYZ in home directory.
XYZ DEF>ABC	File named ABC in UFD named XYZ with password DEF.

File Units (Funits)

A file unit is a logical unit that PRIMOS associates with an open file. A user may have 126 file units open at once. When files are opened by high-level languages other than FORTRAN, the programmer is not aware which file unit number is associated with the file at runtime. Subroutines, however, may be called to open a file with a specified file unit number. (The exact number chosen does not matter as long as it is between 1 and 126.) The file may be accessed through its file unit number. This kind of access may be faster than access by filename, and is more flexible than the file access allowed by the Pascal, PL1G, and PMA languages. A file unit also has a position and an access method, so that when a user reads from a file or writes to the file using the file unit, it is not necessary for the user program to keep track of the file's position and access. Examples of file unit strategy are given with SRCH$$ in this chapter.

Buffer

A buffer is an area of memory addressed by a data name. It is usually defined as an integer array in FORTRAN, and may contain both numbers and characters. It is of variable length, and so is followed by an argument specifying the number of words or characters in buffer.

If separate words or characters of the buffer can be addressed by number, the buffer can be called an array or vector.

Array or Vector

An array is an integer array, with the same characteristics as buffer above. Arrays are sometimes called vectors in this guide.

Home Directory and Current Directory

There is a distinction between home directory and current directory which is made by subroutines, but is not made at PRIMOS command level. For a file management subroutine, the current directory is the one to which the process is currently attached. The home directory, however, is either the one first attached to, or the one defined by a subroutine such as AT$HOM. So that the author of a program may be sure that a process is attached to a certain directory after a series of subroutine calls, including possible failures, routines that handle pathnames always close the specified file unit, then attach to the user's home UFD before attempting any action. If the user's home UFD differs from the current UFD before the call, the process will be attached to the home UFD following the call. In addition, the home directory is the UFD or sub-UFD used as the starting point when the asterisk (*) is used in a pathname by a subroutine call.

Old Partitions

When this chapter refers to old partitions, it means those established under the pre-Rev. 14 file system. Systems that are running under Rev. 18.4 or higher do not support old partitions, so the user can ignore these references.

SUBROUTINE DESCRIPTIONS

The file-manipulation subroutines are described below in alphabetical order. See Table 9-1 for a summary of functions provided.

> **Caution**
>
> Do not omit any arguments in calls to the subroutines described in this section. Do not specify as 0 (or any constant) any arguments returned by the subroutines, such as the error code (integer return code). Always check the error code to see if the subroutine call was successful. It is essential to refer to Appendix D which covers the error-handling scheme for these subroutines.

▶ APSFX$

Purpose

The PL1G subroutine APSFX$ appends a specified suffix to a pathname. It is designed for use with the file-naming convention starting with Rev. 18 that appends standard suffixes to a name by means of a period, such as MYPROG.COBOL. The pathname is checked for the prior existence of the suffix to avoid overwriting an existing file.

Usage

```
DCL APSFX$ ENTRY (CHAR(128)VAR, CHAR(128)VAR, CHAR(32)VAR,
        FIXED BIN);

CALL APSFX$ (in-pathname, out-pathname, suffix, status)
```

in-pathname — Pathname input to check for suffix (128 character maximum).

out-pathname — Pathname returned to caller with desired suffix appended (128 character maximum). 18.1

suffix — This is the suffix to be added to the pathname. It should include the period, and be in capital letters, for example, ".F77" (input; 32 character maximum).

status code — The code returned has the following possible meanings:

-1	Suffix already present, pathname remained untouched.
0	Suffix appended OK.
E$NMLG	Pathname+suffix is more than 128 characters or filename+suffix is longer than 32 characters (FIXED BIN (15)).

Discussion

APSFX$ does not permanently change the name of the file, only the name returned in out-pathname. It is most often used after SRSFX$ is called. After SRSFX$ finds a file and determines its suffix, APSFX$ may add a suffix to the name found.

18.1 APSFX$ is often helpful because SRSFX$ returns two parts to a name — the basename and a suffix. APSFX$ ensures that the name in outpathname has the proper suffix if one is required.

▶ ATCH$$

19

Note

ATCH$$ is obsolete and has been replaced by AT$, AT$ABS, AT$ANY, AT$HOM, AT$OR, and AT$REL.

Purpose

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 that a User File Directory (UFD) is in the Master File Directory (MFD) on a particular logical disk, in a subdirectory in the current UFD, or in the home UFD.

Usage

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

ufdnam — The name of the UFD to be attached (integer array). If key is K$IMFD and ufdnam is the key K$HOME, the home UFD is attached. If the reference subkey is K$ICUR, ufdnam is the name of an array that specifies the name of the UFD to attach to.

namlen — The length in characters (1-32) of ufdnam (INTEGER*2). 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 (INTEGER*2). The parameter ldisk must be a logical disk that is started up. Other values for ldisk 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 integer array containing one of the passwords of ufdnam. passwd can be specified as 0 if attaching to the home UFD. If the reference subkey is K$IMFD or K$ICUR, passwd must be the name of a three-word array that specifies one of the passwords of ufdnam. If passwd is blank, it must be specified as three words, each containing two blank characters.

key — Composed of two subkeys whose values are added together, a REFERENCE subkey and a SETHOME subkey (INTEGER*2). 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$SETH, may be added to the REFERENCE subkey as K$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 0, ldisk is ignored, and it is usually specified as 0.

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

Discussion

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$$ subroutine does not change the home UFD unless the user specifies a change in the subroutine call. The user gets owner status or nonowner status according to the password used. The owner of a file directory can declare, on a per-file basis, what access a nonowner has over the owner's files. The nonowner password may be given only under PRIMOS and PRIMOS III. (Refer to the description of the commands SPAS$$ and SATR$$ in this chapter for more information.)

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

Examples

1. Attach to home UFD:

   ```
   CALL ATCH$$ (K$HOME, 0, 0, 0, 0, CODE)
   ```

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

   ```
   CALL ATCH$$('G.S.PATTON', 10, K$CURR, 'CHARGE', K$ICUR, CODE)
   ```

▶ CNAM$$

Purpose

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

Usage

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

oldnam	The name of the file to be changed (integer array).
oldlen	The length in characters of oldnam (INTEGER*2).
newnam	The new name of the file (integer array).
newlen	The length in characters of newnam (INTEGER*2).
code	An INTEGER*2 variable set to the return code.

Discussion

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, or BADSPT. An E$NRIT error message is generated if this is attempted.

▶ COMI$$

Purpose

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

Usage

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

filnam	The name of the command file to receive the command input stream (integer array). 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 (16-bit integer).
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 (16-bit integer).
code	An integer variable set to the return code (16-bit integer).

▶ COMO$$

Purpose

COMO$$ switches terminal output to file or terminal.

Usage

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

key — A 16-bit 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 integer array containing the name of the file to be opened or 0.

namlen — The length in characters (1-32) of filnam or 0 (16-bit integer).

xx — Reserved. Should be specified as 0 (16-bit integer).

code — Return code from the file system (16-bit integer).

Discusssion

19 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 reenabled or until a unit-127 (command output file) 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 '177, 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 are overwritten. The parameter can be omitted by specifying a pair of blanks or a length of 0.

Error messages (from ERRRTN, ERRPR$) force TTY output on, but leave the command output file open so 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.

The command output unit depends on the FILUNT directive in the CONFIG file at cold start. 18.1

▶ CREA$$

Purpose

CREA$$ creates a new sub-UFD in the current UFD and initializes the new entry. The new sub-UFD is of the same type (ACL or non-ACL) as the current UFD. 19

Usage

```
DCL CREA$$ ENTRY (CHAR NONVARYING(32), FIXED BIN, CHAR NONVARYING(6),
                  CHAR NONVARYING(6), FIXED BIN)

CALL CREA$$ (filnam, namlen, owner-pw, nonowner-pw, code)
```

filnam — The name to be given the new UFD (input).

namlen — The length in characters (1-32) of filnam (16-bit integer).

owner-pw — A six-character array containing the owner password for the new UFD. If opwner-pw(1) = 0, the owner password is set to blanks. owner-pw is ignored if an ACL directory is being created. 19

nonowner-pw — A six-character array containing the nonowner password for the new UFD. If nonowner-pw(1) is 0, the nonowner password is set to zeros. Any password given to ATCH$$ matches a nonowner password of zeros. nonowner-pw is ignored if an ACL directory is being created.

code — A 16-bit integer variable to be set to the return code from CREA$$. Possible values follow.

E$BNAM The supplied name is illegal.

E$BPAR The name length is illegal.

E$EXST An object with the given name already exists.

E$NRIT Add rights were not available on the current directory.

E$WTPR The disk is write-protected.

E$NINF An error occurred, and list rights were not available on the current directory.

19 E$NATT The current attach point is invalid.

Discussion

CREA$$ creates a new subdirectory in the current directory. The new subdirectory is of the same type as its parent. Thus, if CREA$$ is used in an ACL directory, it will create an ACL directory. If used in a password directory it will create a password directory.

Password directories may be explicitly created with the CREPW$ routine. There is no special routine to create ACL directories, since CREA$$ will always create an ACL directory within an ACL directory, and an ACL directory may not have a password directory as its parent.

Passwords can be set such that the password cannot be entered from the keyboard and the directory is accessible only from a program. In any case, passwords can be at most six characters long. Passwords shorter 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 lowercase 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.

19 CREA$$ requires add access on the current UFD.

Example

To create a new UFD with default passwords of blanks for owner and 0 for nonowner:

```
    CALL CREA$$ ('NEWUFD', 6, 0, 0, CODE)
```

▶ FORCEW

Purpose

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.

Usage

```
CALL FORCEW (key, funit [,code])
```

key — Must be 0 (INTEGER*2).

funit — The file unit (1-126) on which a file has been opened (integer array).

code — Standard return code that is E$DISK when a disk error occurred on the file referenced by funit (INTEGER*2). If code is not supplied as an argument, then disk errors will not be reported.

19

Discussion

FORCEW may be used to obtain the status of disk write operations to a file. When a disk write error occurs, all units open on the file are specially marked. When FORCEW is called with the error code parameter included, if an error condition exists, E$DISK is returned and the error mark is reset. If code is not supplied, no action is taken and the error mark is not reset, so it may be sensed at a later time.

Note

The error mark is set in all units associated with the file regardless of which one of them caused the actual error.

▶ GPAS$$

Purpose

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

Usage

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 (integer array).

namlen — The length in characters (1-32) of *ufdnam* (16-bit integer).

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 — A 16-bit integer variable set to the return code.

Discussion

19 GPAS$$ requires protect rights to the current UFD.

Example

To read both passwords of SUBUFD:

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

▶ GPATH$

Purpose

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

Usage

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

key — A 16-bit integer variable specifying the 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).
- K$INIA — Initial attach point (origin).

funit — Specifies file unit number if key is K$UNIT, otherwise ignored (16-bit integer).

buffer — The buffer (data name) where the pathname is to be returned.

bufflen — Specifies maximum buffer length in characters (16-bit integer). 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 (16-bit integer).

code — Return code (16-bit integer). Possible values are:

- 0 — 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$NATT Not attached to any UFD (keys K$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.

Examples

The following are examples of information returned as the result of using GPATH$. The lowercase names define what information the examples (in uppercase) 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$

Purpose

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

Usage

log = NAMEQ$ (filnam1,namlen1,filnam2,namlen2)

filnam1 The first filename for comparison (integer array).

namlen1 The length in characters of filnam1 (16-bit integer).

filnam2 — The second filename for comparison (integer array).

namlen2 — The length in characters of filnam2 (16-bit integer).

Discussion

NAMEQ$ performs a character-by-character comparison of filnam1 and filnam2 for the length of namlen1 or namlen2, whichever is shorter. The names supplied must be valid filenames.

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

▶ PRWF$$

Purpose

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

Usage

CALL PRWF$$ (rwkey+poskey+modekey, funit, LOC(buf), nw, pos, rnw, code)

rwkey — This INTEGER*2 key, which cannot be omitted, indicates the action to be taken. Possible values are:

K$READ — Read nw words from funit into buf.

K$WRIT — Write nw words from buf 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 — An INTEGER*2 key indicating the positioning to be performed (if omitted, same as K$PRER). Possible values are the following.

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.

modekey An INTEGER*2 key that may be used to transfer all or a convenient number of words (if omitted, read/write nw). Possible values are:

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

K$FRCW Perform a 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$$ or by a PRIMOS command. PRWF$$ actions are performed on this file unit.

LOC(buf) The data buffer to be used for reading or writing. If buffer is not needed, it can be specified as INTL(0).

nw The number of words to be read or written (mode=0) or the maximum number of words to be transferred (mode=K$CONV). nw may be between 0 and 65535 (INTEGER*2).

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 (INTEGER*2).

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

Discussion

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 0 can be generated by specifying 000000 or INTL(0) in FTN, 0L in PMA or Pascal.

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 is positive. Positioning takes place before or after the data transfer, depending on the key. If nw is 0 in any of the calls to PRWF$$, no data transfer takes place, and PRWF$$ performs a pointer position operation.

The modekey subkey of PRWF$$ is most frequently used to transfer a specific number of words on a call to PRWF$$. In these cases, the modekey is 0 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$$. The number of words transferred is a number convenient to the system, and therefore speeds up program runtime. The number of words actually transferred is set in rnw. For examples of PRWF$$ use in a program, refer to the file-manipulation examples in Chapter 5.

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 (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 key K$FRCW later in this chapter.

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$$, PRIMOS maintains a file pointer for every open file. When a file is opened by a call to SRCH$$, 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 value is 0, 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 wraparound 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 ten 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 ten 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 that is open on unit number UNIT to be positioned to the tenth word in the file, and the first ten 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 performed immediately.

6. The next program reads and writes SAM and DAM files using PRWF$$.

```
/*********************************************************************/
/* Copy SAM and DAM files                                            */

cp$$fl:
     proc(sunit, tunit, err_info, code);

%include 'syscom>keys.pl1';
%include 'syscom>errd.pl1';

%replace maxsiz     by 1024;            /* maximum record size in words */

dcl  sunit           fixed binary(15), /* unit source file is open on  */
     tunit           fixed binary(15), /* unit target file is open on  */
     err_info        fixed binary(15), /* if code ^= 0, indicates which
                                       /* file caused error;1 = source,*/
                                       /* 2 = target                   */
     code            fixed binary(15); /* standard error code          */
dcl  recbuf(maxsiz)  fixed binary(15); /* I/O buffer                   */
dcl  words_read      fixed binary(15); /* actual words read by prwf$$  */
dcl  words_written   fixed binary(15); /* actual words written by prwf$$*/
dcl  eof             bit(1);
dcl  recbuf_ptr      pointer options(short);
dcl  addr            builtin;
dcl  errpr$          entry(bin, bin, char(*), bin, char(*), bin);
dcl  user_proc       entry;

dcl  prwf$$          entry (fixed binary(15),
                                        /* keys (rwkey+poskey+mode)     */
                     fixed binary(15),  /* unit to perform action on    */
                     pointer options(short),
                                      /* address of data buffer      */
                    fixed binary(15), /* words to read or write      */
                    fixed binary(31), /* position value              */
                    fixed binary(15), /* actual words read or written*/
                    fixed binary(15)); /* standard error code        */

/**********************************************************************/

err_info = 0;
code = 0;
recbuf_ptr = addr(recbuf);
eof = '0'b;

do while (^eof);
   call prwf$$(k$read, sunit, recbuf_ptr, maxsiz, 0, words_read,
               code);
   if code ^= 0
      then if code ^= e$eof
              then do;
                   err_info = 1;
                   return;
                   end;
              else eof = '1'b;
a:
  call prwf$$(k$writ,tunit,recbuf_ptr,words_read,0,words_written,code);
  if code ^= 0
      then if code = e$dkfl
              then do;
                   call errpr$(k$irtn, code, '', 0, 'cp$$fl', 6);
                   call user_proc;        /* Wait for response */
                   go to a;
                   end;
               else do;
                   err_info = 2;
                   return;
                   end;
end;
return;
end cp$$fl;
/**********************************************************************/
```

More examples of the use of PRWF$$ are given with the file-system examples in Chapter 5.

▶ Q$READ

Purpose

This routine returns information about quota counters and the time-record product of disk record usage for the current quota UFD. These concepts are explained in the System Administrator's Guide.

Usage

```
DCL Q$READ ENTRY (CHAR(128)VAR, FIXED BIN (31), FIXED BIN, FIXED BIN
                FIXED BIN)

CALL Q$READ (pathname, quota-info, max-entries, type, code)
```

pathname — Name of the directory whose quota information is to be read (input). List access must be available either on the directory itself or on its parent. If pathname is null, information for the current directory is returned.

quota-info — An array returning the quota information:

- quota-info(1) Data size of disk record (440 or 1024 words). 19
- quota-info(2) Directory records used.
- quota-info(3) Max number of records of quota (0 if nonquota).
- quota-info(4) Total records used.
- quota-info(5) Time-record product (computed in record-minutes) (0 if nonquota).
- quota-info(6) Date/time last updated (0 if nonquota).

  Date format is word one: YYYYYYYMMMMDDDDD.

  Time is word two (seconds since midnight divided by four).
- quota-info(7) Reserved for future use.
- quota-info(8) Reserved for future use.

max-entries — Number of entries in quota-info (input).

type	Type of directory (input):	
	0	Quota Directory
	1	Non-quota Directory
code	Standard return code:	
	E$NINF	Insufficient access to read quota.

Discussion

When this call is invoked on a nonquota directory, the arguments detailed below will have the following information returned. The type will be 1 and quota-related information (max, time-record product, and date/time) will be 0. Directory records used will indicate the sum of the records used by the files in that directory plus the records used by the directory file itself. Total records used will indicate the sum of the records used for all files inferior to this directory mode.

Quota directories will return a type equal to 0, and all of the quota information. Directory records used and total records used will be the same as in the nonquota directory case. 19

The routine will enter as many values into the array buf as is specified by buflen, up to a maximum of eight. Entries which are reserved for future use will have an undefined value.

Use of the Accounting Meter Returned by Q$READ

The system keeps an accounting usage meter in each quota directory. This meter is a summation of the time intervals that each disk record has been in use.

The accounting meter is a counter that acts as an unsigned number, which is to say that it counts to all ones and then goes to 0. The system also indicates when the last update occurred.

The calculation used is given below. The USAGE is computed in record-minutes.

```
TIME = (Current date/time) - (Date/time quota last modified)
USAGE = USAGE + (Records used ) * TIME
```

An accounting program would use a similar algorithm to calculate the current record-time product.

▶ Q$SET

Purpose

This routine sets a maximum quota on a SUBUFD in the current directory. If the named directory is not already a quota directory, it will become one.

Usage

```
DCL Q$SET ENTRY (FIXED BIN, CHAR(128)VAR, FIXED BIN (31), FIXED BIN)

CALL Q$SET (key, pathnam, max-quota, code)
```

key — Must be K$SMAX (set maximum quota) (input).

pathname — An array containing the name of the sub-UFD to receive the quota (input). Protect access must be available on the directory's parent. 19

max-quota — Maximum quota for the directory and its subtree (input). If this is 0, any existing quota is removed.

code — Standard return code:

E$NRIT	Insufficient access to set quota.
E$IMFD	Quota not permitted on MFD.
E$QEXC	Used records greater than new maximum (WARNING).
E$FIUS	Directory in use during attempt to convert from nonquota to quota.

▶ RDEN$$

Purpose

RDEN$$ positions in or reads from a UFD.

19 **Note**

For Pascal and PL1G programmers, RDEN$$ is obsolete and has been replaced with DIR$RD and ENT$RD.

Usage

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

key — A 16-bit 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$FNTF is returned. If namlen is 0, the next entry is returned.
- K$GPOS — Return the current position in the UFD as a 32-bit integer in filnam.
- K$UPOS — Set the current position in the UFD from the 32-bit integer in filnam. This key should be used only with a position of 0.
- K$POSN — Return access category entries.

funit — A unit on which a UFD is currently opened for reading (INTEGER*2). (A UFD may be opened with a call to SRCH$$.)

buffer — A one-dimensional array into which entries of the UFD are read. If the key is 3, the first word of buffer will have bit 1 set on if the object is not default-protected.

buflen — The length, in words, of buffer (INTEGER*2).

rnw — An INTEGER*2 variable that will be set to the number of words read.

filnam — An INTEGER*4 variable used for keys of K$GPOS and K$UPOS, or a name (character string) for use with K$NAME.

namlen — An INTEGER*2 variable specifying the length in characters (0-32) of filnam. This variable is only used with K$NAME.

code — An INTEGER*2 variable to be set to the return code:

E$FNTF	The entry is not in the directory.
E$EOF	No more entries.
E$BFTS	Buffer is too small for the entry.

Discussion

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.

> **Caution**
>
> Directory positioning is obsolete and should not be necessary.

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 in Figure 9-1 and discussed below for Revs before 19. (For Rev. 19 format, see DIR$RD.) All numbers are decimal unless preceded by a colon (:). 19

```
    ________
 0 |  ECW   |  Entry Control Word (type/length)
 1 |F       |
   | I      |
   |   L    |
   |     E  |
   |  ...   |  Filename (blank-padded)
   |N       |
   | A      |
   |   M    |
   |_____E__|
17 | PROTEC |  Protection (owner/nonowner)
18 |RESERVED|  Reserved for future use
19 | FILTYP |  File type  <--- (end of entry for type=1)
20 | DATMOD |  Date last modified
21 | TIMMOD |  Time last modified
22 |RESERVED|  Reserved for future use
23 |RESERVED|  Reserved for future use
    ‾‾‾‾‾‾‾‾
```

File Entry Format
Figure 9-1

ECW — Entry Control Word. An ECW is the first word in any entry and consists of two 8-bit subfields. The high-order eight bits indicate the type of the entry, the low-order eight 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.

:001030 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 eight bits, the nonowner in the low-order eight 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:

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD

On an old partition, the file type is invalid. The file must be opened with SRCH$$ to determine its type.

Of the high-order eight 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 read/write lock. (See below.)

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, and 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.

The 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
0	0,0	Use system-wide RWLOCK to regulate concurrent access.
1	0,1	Allow arbitrary readers or one writer.
2	1,0	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 0.

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 0. 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 0.

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 0, so the system-wide RWLOCK setting continues to control access to such files.

The COPY command with the -RWLOCK option copies the per-file read/write lock setting along with the file. 19

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)
```

3. This program reads directory entries sequentially using RDEN$$.

```
/*****************************************************************/
rd$dir:
     proc(dunit, rden_ptr, code);

dcl  dunit              bin,      /* unit directory is open on */
     rden_ptr           pointer,  /* pointer to rden_buffer    */
     code               bin;      /* standard error code       */

%include 'syscom>keys.pl1';
%include '*>insert>parameters.ins.spl';

dcl  rden$$             entry(bin,bin,(24)bin,bin,bin,char(*),
                              bin, bin),
     rden_buffer(24)    bin based(rden_ptr),
     rden_name_ext        char(32) defined rden_buffer(2),
     rden_name_local      char(32);
dcl  i                    bin;
dcl  trim                 builtin;

/******************************************************************/

 call rden$$(k$read, dunit, rden_buffer, 24, i, '', 0, code);

 rden_buffer(23) = rden_buffer(19); /* Copy non_default_acl bit*/
 rden_buffer(19) = rden_buffer(18); /* Copy protection keys    */
 rden_name_local = rden_name_ext;   /* Copy name for trim (Since
                                            the strings overlap). */
 rden_ptr -> rden_buffer_.filename = trim(rden_name_local,'01'b);
 return;
 end rd$dir;    /* rd$dir */
/******************************************************************/
```

4. The next example reads directory entries by name using RDEN$$.

```
/******************************************************************/
rd$ent:
     proc(treename, rden_ptr, code);

dcl  treename    char(128) var, /* file info is wanted for    */
     rden_ptr    pointer,       /* pointer to rden_buffer     */
     code        bin;           /* standard error code        */

%include 'syscom>keys.pl1';
%include '*>insert>parameters.ins.spl';

dcl  rden$$              entry(bin, bin, (24) bin, bin, bin, char(*),
                               bin, bin),
   rden_buffer(24)       bin based(rden_ptr),
   rden_name_ext         char(32) defined rden_buffer(2),
   rden_name_local       char(32);
dcl  srch$$              entry(bin, bin, bin, bin, bin, bin);
dcl  tatch$              entry(char(*) var, bin);
dcl  path$               entry(char(*) var) returns(char(128) var);
dcl  entry$              entry(char(*) var) returns(char(32) var);
dcl  home$               entry();
dcl  close$              entry(bin);
dcl  (i,
     icode,
     unit)               bin;
dcl  tree                bit(1) aligned,
     filename            char(32) var;
dcl  (length,
     trim,
     addr,
     index)              builtin;

/********************************************************************/
```

```
tree = (index(treename, '>') ^= 0);
if tree
   then do;
        call tatch$(path$(treename), code);
        if code ^= 0
           then go to clean_up;
        end;

call srch$$(k$read + k$getu, k$curr, 0, unit, i, code);
if code ^= 0
   then go to clean_up;

filename = entry$(treename);
call rden$$(k$name, unit, rden_buffer, 24, i, (filename),
            length(filename), code);

call close$(unit);

rden_buffer(23) = rden_buffer(19); /* Copy non_default_acl bit */
rden_buffer(19) = rden_buffer(18); /* Copy protection keys     */
rden_name_local = rden_name_ext;   /* Copy name for trim (Since
                                      the strings overlap).    */
rden_ptr -> rden_buffer_.filename = trim(rden_name_local, '01'b);

clean_up:
     if tree
        then call home$;
     return;

     end rd$ent;
```

RDLIN$

Purpose

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

Usage

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

funit — A file unit (1-126) on which the file to be read is open (INTEGER*2).

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

count — The size of buffer in words (INTEGER*2).

code — A return variable set to 0 for no errors, or to an error code for an error (INTEGER*2). See PRWF$$ for a list of possible error codes.

Discussion

A line of characters from funit is read into buffer, two characters per word. Lines on the disk are separated by the NEWLINE character. Compressed files are treated this way: the character DC1 (221 octal) followed by a count when read from the disk is replaced by that many blanks.

If the line on the disk is less than 2*count characters, the remaining space in buffer is filled with blanks. If the line on the disk is greater than 2*count characters, only 2*count characters fill buffer and the remaining characters on the disk file line are ignored. In all cases, the NEWLINE never appears as part of the line in buffer.

RDLIN$ is the same routine as I$AD07 except that the altrtn argument has been replaced by the code argument.

▶ REST$$

Purpose

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

Usage

```
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 executable image (integer array).

namlen — The length in characters (1-32) of filnam (INTEGER*2).

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

Note

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

► RESU$$

Purpose

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

Usage

```
CALL RESU$$(filnam, namlen)
```

filnam — The name of the file containing the code.

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

Discussion

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

▶ SATR$$

Purpose

SATR$$ allows the setting or modification of an object's attributes in its UFD entry. The attributes that may be set include:

- Password protection
- Date/time modified
- 19 Dumped bit
- Read/write lock
- Delete-protect switch

Usage

CALL SATR$$ (key, object, namlen, attributes, code)

key

A 16-bit integer variable specifying the action to take. Possible values are:

K$PROT Set password protection attributes from attributes(1). attributes(2) is ignored for old partitions and must be 0 for new partitions. (It is reserved for expansion.) The meaning of the protection bits in attributes(1) is given under the description of RDEN$$.

K$DTIM Set date/time modified from attributes(1) and (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 Set the read/write lock on a per-file basis. Bits 15 and 16 of attributes(1) are set by the user for specific lock values. Refer to RDEN$$ for further information on the read/write lock.

K$SDL Set the delete switch (for use with ACLs). If attributes(1) is not 0, the delete switch is set. If attributes(1) is 0, the switch is cleared. 19

Note

The date/time modified and the dumped bit are changed by PRIMOS. When PRIMOS changes these fields for a file, the corresponding fields of the file's parent UFD are 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.

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

object The name of the object (file or other item) whose attributes are to be modified. The current UFD is searched for object (CHAR NONVARYING(32)). 19

namlen The length in characters of filnam (16-bit integer).

attribute Field containing the attributes; variable, depending on key:

- For K$PROT, a 16-bit structure defining the password protection rights for the object. This structure is defined below.
- For K$DTIM, a 32-bit structure containing the date/time to set in FD standard format, which is described below.
- For K$DMPB, this field is ignored.
- For K$RWLK, one of the following sub-keys as a FIXED BIN(15): 19

K$DFLT	Use system default value.
K$EXCL	Unlimited readers OR one writer.
K$UPDT	Unlimited readers AND one writer.
K$NONE	Unlimited readers and writers.

- For K$SDL, a 16-bit quantity. If nonzero, the delete-protect switch is set on. If zero, it is set off.

```
/* do copies                     */

          if type < 2
             then call cp$$fl(sfunit, tfunit, err_info, code);
             else call cp$$sd(sfunit, tfunit, err_info, code);

/* close the entries just copied */

          call srch$$(k$clos + k$iseg, sunit, 0, sfunit, trash,
                      tcode);
          call srch$$(k$clos + k$iseg, tunit, 0, tfunit, trash,
                      tcode);
          if code ^= 0
             then return;
          end;
     end;
err_rtn_1:
     err_info = 1;
     return;
err_rtn_2:
     err_info = 2;
     return;
     end cp$$sd;
```

▶ SPAS$$

Purpose

SPAS$$ sets the passwords of the current UFD.

Usage

CALL SPAS$$(owner-pw, nonowner-pw, code)

owner-pw   A six-character array that contains the password to set as the owner password.

nonowner-pw   A six-character array that contains the password to set as the nonowner password.

code   A 16-bit integer variable set to the return code.

code — A 16-bit integer variable set to the return code:

E$BKEY — An illegal key value was passed.

E$BNAM — Object name is illegal.

E$BPAR — namlen is less than 0 or greater than 32.

E$NATT — The current attach point is invalid.

E$NRIT — Protect access (delete access for K$SDL) was missing from the current directory.

E$WTPR — The disk is write-protected.

E$NINF — An error occurred during search of the directory, and list access was not available.

E$FNTF — The object does not exist.

19 E$IACL — The object was an access category, and a key other than K$DTIM was used.

E$DIRE — The object was a directory, and the K$RWLK key was used.

Discussion

The password protection structure is as follows:

```
dcl 1 pw_protection,
      2 owner_rights,
        3 ignored bit(5),
        3 delete bit(1),
        3 write bit(1),
        3 read bit(1),
      2 non_owner_rights,
        3 ignored bit(5),
        3 delete bit(1),
        3 write bit(1),
        3 read bit(1);
```

The standard FS-format date structure is:

```
dcl 1 fs_date,
      2 year bit(7),
      2 month bit(4),
      2 day bit(5),
      2 quadseconds fixed bin(15);
```

The meaning of these elements is:

year — Year modulo 100, with the exception that years 100-128 mean 2000-2028.

month — Month, from 1 for January to 12 for December. 19

day — Day of the month, from 1 to 31.

quadseconds — Number of quadseconds (groups of four seconds) elapsed since midnight of the date described by the three preceding fields.

Note

SATR$$ does not check the validity of the supplied date and time. Users must assure that the date/time passed is legal.

Owner rights are required on the UFD containing the entry to be modified, except with K$SDL, which requires delete access.

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').

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 nonowner attributes to read-only (note carefully the 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$$

Purpose

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

Usage

```
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 code (integer array).

namlen — The length in characters (1-32) of filnam (16-bit integer).

code — A standard return code (16-bit integer).

▶ SGDR$$

Purpose

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

Usage

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

key — A 16-bit integer specifying the action to be performed. Possible values are:

- K$SPOS — Move the file pointer of funit to the position given by the value of entrya. Return 1 in entryb if entrya contains a file, return 0 if entrya exists but does not contain a file, return -1 if entrya does not exist (is beyond EOF). If EOF is reached on K$SPOS, the file pointer is left at EOF. The directory must be open for reading or both reading and writing.

- K$FULL — Move the file pointer of funit to the position given by the value of entrya. If the position contains a file, set entryb to the value of entrya. If the position is empty, search for the first nonempty entry following the position specified. If a nonempty entry exists, set entryb to the position of that entry. If the EOF is reached and an entry with a file has not been found, then return -1 in entryb. If EOF is reached on K$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, or 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 (16-bit integer).

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 A 16-bit integer variable set to the return code, according to the key used.

Discussion

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

A K$MSIZ call with entrya equal to 0 causes the directory to have no entries. If the value of entrya is such that it truncates 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 a directory is read sequentially (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, that is, the entry following the first K$POS to return -1 in entryb.

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$$ (K$GOND, 1, 0, SIZE, CODE)
IF (CODE.NE.0) GOTO <error handler>
CALL SGDR$$ (K$MSIZ, 2, SIZE, 0, CODE)
```

3. This program reads and writes segment directories using SGDR$$.

```
/**********************************************************/
cp$$sd:
    proc(sunit, tunit, err_info, code) recursive;

%include 'syscom>keys.pl1';
%include 'syscom>errd.pl1';

dcl  sunit       fixed bin(15),
     tunit       fixed bin(15),
     err_info    fixed bin(15),
     code        fixed bin(15);

dcl (entrya,
     entryb,
     entry_no)   fixed bin(15);
dcl (sfunit,
     tfunit)     fixed bin(15);
dcl (newfil,
     trash,
     tcode,
     rtnval,
     type)       fixed bin(15);

dcl  errpr$      entry(bin, bin, char(*), bin, char(*), bin);
dcl  srch$$      entry(bin, bin, bin, bin, bin, bin);
dcl  cp$$fl      entry(bin, bin, bin, bin);
                 /* cp$$fl is defined in example 6 for PRWF */
dcl  sgdr$$ entry /*read segdir entries*/ (fixed binary(15),
                               /* key */
                 fixed binary(15), /* unit on which segdir is
                               /*open*/
                 fixed binary(15), /* entrya */
                 fixed binary(15), /* entryb */
                 fixed binary(15)); /* standard error code */

set_target_size:            /* make target segdir same number
                            /* of entries as source */
     err_info = 0;
     call sgdr$$(k$gond, sunit, entrya, entry_no, code);
     if code ^= 0
        then go to err_rtn_1;
     call sgdr$$(k$msiz, tunit, entry_no, entryb, code);
     if code ^= 0
        then go to err_rtn_2;

main_loop:

  do entry_no = 0 repeat (entry_no + 1);

/* position segdirs                                         */
     call sgdr$$(k$spos, sunit, entry_no, rtnval, code);
     if code ^= 0
        then go to err_rtn_1;
     if rtnval < 0
        then return;                  /* end of file        */
     call sgdr$$(k$spos, tunit, entry_no, entryb, code);
     if code ^= 0
        then go to err_rtn_2;
     if entryb < 0
        then do;
             call errpr$(k$irtn, e$null, 'Unrecoverable
                        error', 19, 'cp$$sd', 5);
             stop;
             end;

     if rtnval = 1
        then do;

/*found a nonnull entry in source,  */
/*     open it and same entry in target*/

          call srch$$(k$read + k$iseg + k$getu, sunit, 0,
                      sfunit, type, code);
          if code ^= 0
             then go to err_rtn_1;
          newfil = k$nsam;
          if type = 1
             then newfil = k$ndam;
          if type = 2
             then newfil = k$nsgs;
          if type = 3
             then newfil = k$nsgd;
          call srch$$(k$rdwr+k$iseg+k$getu+newfil, tunit, 0,
                      tfunit, trash, code);
          if code ^= 0
             then do;
                  call srch$$(k$clos + k$iseg, sunit, 0,
                              sfunit, trash, tcode);
                  go to err_rtn_2;
                  end;
```

Discussion

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

Passwords which are not intended to be typed from the terminal but accessed through programs only can have any bit pattern.

▶ SRCH$$

Purpose

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

Note

19 At Rev. 19, the delete functions of SRCH$$ are handled by FIL$DL and SGD$DL.

Usage

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

action — A 16-bit 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.

K$DELE — Delete file filnam.

K$EXST — Check on existence of filnam.

ref — A 16-bit key modifying the action key 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. (This is the PRIMOS file unit, not the FORTRAN unit.) The unit number is returned in funit. When this key is used, SRCH$$ supplies a unit number not currently in use. See example 6 below for use of this key.

newfil — A 16-bit key 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.

Note

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

filnam — Name of the file to be opened (integer array, two 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 126 (1 to 15 under PRIMOS II) on which a segment directory is already open.

namlen — The length in characters (1-32) of filnam (16-bit integer).

funit — The number (1-15 under PRIMOS II, 1-126 under PRIMOS) of the file unit to be opened or closed, or returned argument with K$GETU key (16-bit integer).

type A 16-bit 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:

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD

code An integer variable set to the return code.

Discussion

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$AD07, O$AD07, RDASC, and WRASC. Information is also transferred through the I/O statements in all languages.

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 command or the SATR$$ subroutine. These access rights (read, write, delete, 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 Lock

Under default conditions, the system allows any number of readers 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. These locks also hold for a single user attempting to open a file on multiple file units. If the lock is violated, the FILE IN USE error code is generated (E$FIUS).

This lock 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$$ attempts to close by filnam unless filnam is specified as 0, in which case it closes the file unit specified. If filnam is not found, an error is generated (code = E$FNTF), but if the file unit is specified, SRCH$$ 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 lock 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 zero 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 lock 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. (The 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 FIX_DISK maintenance program is run by the system operator. See the System Administrator's Guide.

Checking the Existence of a File

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

Operations on Files That Are UFDs

Files in the current UFD that are sub-UFDs can be opened only for reading. The contents of entries of sub-UFDs can be read through calls to RDEN$$ and GPAS$$ once the sub-UFD 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 DELETE command can delete a nested structure of UFDs, provided they are not protected.

Operations Involving Segment Directories

Segment directories are directories in which the files are referenced by their position in the directory rather than by 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, use SRCH$$ and SGDR$$ to open the segment directory and position to the desired entry 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, and 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 0 using SGDR$$, close the directory, and then delete it with SRCH$$. The DELETE subcommand of the SEG command may be used to delete a segment directory.

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 and Pathnames

For a discussion of filenames and pathnames, see the introduction to this chapter.

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$$(K$CLOS, 0, 0, DMUNIT, 0, CODE)
   CALL SRCH$$ (K$DELE+K$ISEG, SGUNIT, 0, 0, 0, CODE)
   ```

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

   ```
   CALL SRCH$$ (K$EXST+K$IUFD, 'MY.BLACK.HEN', 12, 0, 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, 0, 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, CODE)
    IF (CODE .NE. 0) 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 0 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 user process has exceeded the maximum number of file units allowed, or the total number of file units in use for all processes exceed the maximum number of file units available.

7. Open file by name.

   ```
   /******************************************************/

   open$:
       proc(key, treename, unit, type, code);

   %include 'syscom>keys.pl1';

   %replace sam_file     by 0,
            dam_file     by 1,
            sam_segdir   by 2,
            dam_segdir   by 3,
            directory    by 4;

   dcl  key          bin,
        treename     char(128) var,
        unit         bin,
   ```

```
     type          bin,
     code          bin;
dcl  srch$$        entry(bin, char(*), bin, bin, bin, bin),
     newfil        bin;
dcl  tatch$        entry(char(*) var, bin);
dcl  path$         entry(char(*) var) returns(char(128) var);
dcl  entry$        entry(char(*) var) returns(char(32) var);
dcl  home$         entry();
dcl  tree          bit(1) aligned,
     filename      char(32) var;
dcl (length,
     index)        builtin;

/************************************************************/

     code = 0;
     tree = (index(treename, '>') ^= 0);
     if tree
        then do;
             call tatch$(path$(treename), code);
             if code ^= 0
                then go to clean_up;
             end;

     filename = entry$(treename);

     newfil = k$nsam;
     if key = k$writ | key = k$rdwr
        then if type = dam_file
                then newfil = k$ndam;
             else if type = sam_segdir
                then newfil = k$nsgs;
             else if type = dam_segdir
                then newfil = k$nsgd;

     call srch$$(key+newfil+k$getu,(filename),length(filename),
                 unit, type, code);

clean_up:
     if tree
        then call home$;
     return;

     end open$;
```

▶ SRSFX$

Purpose

The subroutine SRSFX$ searches for a file according to the filenaming standards of Rev. 18 and higher. The caller supplies a list of possible suffixes.

Usage

```
DCL SRSFX$ ENTRY (FIXED BIN, CHAR(*)VAR, FIXED BIN, FIXED BIN,
                  FIXED BIN, CHAR(32)VAR, CHAR(32)VAR,
                  FIXED BIN, FIXED BIN)
                  [ RETURNS(FIXED BIN(31)); ]

CALL SRSFX$ (key, pathname, unit, type, n-suffixes, suffix-list,
             basename, suffix-used, status);

chrpos = SRSFX$ (key, pathname, unit, type, n-suffixes,
                 suffix-list, basename, suffix-used, status);
```

key — Key(s) to use for the search — same as for SRCH$$ (input).

18.1 pathname — Pathname to use for search (remains unchanged) (input).

unit — File unit opened (returned with K$GETU) or file unit to use for SRCH$$ action without K$GETU (input).

type — File type opened (output).

n-suffixes — Number of suffixes in suffix-list (input). A value of 0 indicates not to use the file-naming standards with suffixes for the search.

suffix-list — List of desired suffixes to use (input). Each suffix should include the period and be in capital letters, for example, suffix-list(i) = ".F77".

basename — This is the base filename, that is, without a suffix according to the suffix-list. This is useful to callers who want to append a different suffix to the base filename. For example, FTN PROG.TEST.FTN would produce output files with "PROG.TEST" as the basename used, such as "PROG.TEST.BIN" (output).

suffix-used — This is the index, in the suffix-list given, of the suffix used for the search. As mentioned, a value of 0 denotes that the null suffix was used (output).

status	Status returned from the search operation (same as for APFSX$).
chrpos	When SRSFX$ is used as a function call, this is returned. The first word points one character past the pathname component that caused the error. The second word is the pathname length.

Discussion

SRSFX$ is intended for use with the filenaming convention, starting with Rev. 18, that appends a standard suffix by means of a period, as in MYPROG.PASCAL. The suffix list defines both the suffixes to scan for and the search order. If the suffix already exists at the end of the filename, then a tree search is performed with the pathname as is.

If none of the desired suffixes are found, a tree search is performed in the following manner: the subroutine attaches to the appropriate directory, each suffix in the list is appended to the filename, and a search is done. In this way the suffix list defines the search order. The routine returns when a "filename.suffix" is found or the suffix list is exhausted.

If a file is found, the index (in the suffix list) of the last suffix in the filename is returned; if no file is found, or if none of the suffixes in the list is on the found filename, an index of 0 is returned. 18.1

SRSFX$ can be combined with APSFX$ to force a name to have a suffix according to the current filenaming conventions, even if the file did not originally have one. For example, the ACL command SET_ACCESS looks for an access category with the suffix .ACAT. If SRSFX$ finds a file with no such suffix, APSFX$ may then be used to return the filename plus the suffix required for the next step.

Restrictions:

- The null string is not allowed as an element of the suffix list. The null suffix is assumed if no desired suffix is found. In this case the suffix index is set to 0 and a processor may then choose to use the old prefix conventions B_, L_, etc., for its output files.
- If the suffix-list contains ".F77", a pathname such as "pathname>.F77" will be treated as a valid suffix found, i.e., ".F77". The filename returned will be '', the null string.

- If the filename + suffix exceeds 32 characters or the pathname + suffix exceeds 128 characters, a search with suffix will not be done and the next suffix is attempted. For example, 18.1 a filename of 32 characters will simply be searched for as is.

- The suffixes in the suffix list provided by the caller must contain the period and be all capital letters, for example, ".F77".

▶ TSRC$$

18.1 **Note**

TSRC$$ is obsolete and has been replaced with SRSFX$.

Purpose

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

Usage

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

action A 16-bit key indicating the action to be performed. Possible values are:

K$READ Open pathname for reading on funit.

K$WRIT Open pathname for writing on funit.

K$RDWR Open pathname for reading and writing on funit.

K$DELE Delete file pathname.

K$EXST Check on existence of pathname.

K$CLOS Close pathname (not funit).

K$GETU Open pathname on an unused file unit selected by PRIMOS. The unit number is returned in funit.

newfil — A 16-bit key indicating the type of file to create if pathname 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.

pathname — An array specifying a file in any directory or subdirectory, packed two characters per word.

funit — The number (1-126) of the file unit to be opened or deleted (16-bit integer). funit is closed before any action is attempted.

chrpos — A two-element integer array for character position set up as follows:

chrpos(1) On entry, set to contain the position in the array pathname occupied by the first character of the filename. (The count starts at 0.) On exit, it will be pointing one past the last character that was part of the pathname. 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 pathname component that caused the error. chrpos(1) is always modified by this subroutine, so it must be set up before each call.

chrpos(2) The number of characters in the pathname array (16-bit integer).

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:

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD

code — A 16-bit integer variable set to the return code. If no errors, code is 0.

Caution

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

▶ UPDATE

Purpose

Under PRIMOS II, this subroutine updates the current UFD.

Usage

CALL UPDATE (key, 0)

key	Value must be 1 to update current UFD, send DSKRAT buffers to disk, if necessary, and undefine DSKRAT in memory (INTEGER*2).

Discussion

This call is effective only under PRIMOS II. Under PRIMOS III or PRIMOS it has no effect.

▶ WTLIN$

Purpose

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

Usage

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

funit	A file unit (1-126) on which the file to be written is open for writing (16-bit integer).
buffer	An integer array of count words from which the line of characters is to be written. It should contain two characters per word.

count	The size of buffer in words (16-bit integer).
code	A 16-bit return code.

Discussion

Information is written on the disk in compressed ASCII format. Multiple blank characters are replaced by the character DC1 (221 octal) followed by a character count. Trailing blanks are removed and the end of record is indicated by adding a NEWLINE character, or a NEWLINE character followed by null. WTLIN$ is the same routine as O$AD07, except that the altrtn argument has been replaced by the code argument.

10
System Subroutines

This chapter describes subroutines that perform PRIMOS system functions. For explanations of the argument names used (such as funit), see Chapter 2.

Table 10-1 summarizes the functions available.

▶ BREAK$

Purpose

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

Usage

CALL BREAK$ (logic-value)

logic-value	A 16-bit integer whose value can be 1 for .TRUE. or 0 for .FALSE. (LOGICAL).

Table 10-1
Operating System Subroutines

19

Phantom Management	
PHANT$	Start a phantom (obsolete).
PHNTM$	Start a phantom (same login name only).
LON$CN	Enable or disable logout notification.
LON$R	Retrieve logout notification information.
Read or Write	
C1IN$	Get one character from command file or terminal.
CL$GET	Read a line of text from command file or terminal.
CNIN$	Move characters.
COMANL	Read a line of text.
GCHAR	Get a character from an array.
SCHAR	Store a character in an array.
Error Checking	
CL$PIX	Parse a command line.
ERRPR$	Interpret a return code.
RDTK$$	Parse a command line.
Manage User Environment	
BREAK$	Inhibit or enable CONTROL-P.
DUPLX$	Return terminal configuration word.
ERLK$$	Read or set erase and kill characters.
EXIT	Return to PRIMOS.
GINFO	Check operating system being used.
GV$GET	Retrieve the value of a global variable.
GV$SET	Set the value of a global variable.
LOGO$$	Log out a user or process.
RECYCL	Pass control to next user.
TIMDAT	Return system and user information.
Manage File Access	
FNCHK$	Check a filename for valid format.
IDCHK$	Check an id for valid format.
PWCHK$	Check a password for valid format.
TEXTO$	Check a filename for valid format (obsolete).
TNCHK$	Check a pathname for valid format.

Discussion

The LOGIN command initializes the user terminal so that the CONTROL-P or BREAK key causes an interrupt (QUIT). Under PRIMOS III and PRIMOS, the BREAK$ routine, if 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 master list of the QUIT status for each user. Each call to BREAK$ to inhibit or enable QUIT increments or decrements a counter, respectively. QUITs are enabled only when the counter is 0; the counter goes positive with inhibits and cannot be decremented below 0.

Under PRIMOS II, BREAK$ has no effect.

▶ C1IN$

Purpose

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

Usage

```
CALL C1IN$(char)
```

Discussion

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

Line feeds are discarded by the operating system, and are not detected by the C1IN subroutine.

▶ CL$GET

Purpose

CL$GET reads a single line of input text from the currently defined command input stream (terminal or command file). 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.

Usage

CALL CL$GET (comline, comlinesize, status)

comline — Varying character string into which the text will be read from the command input stream (CHARACTER(*) VAR).

comlinesize — Maximum length, in characters, of comline. Because comline is a varying string, it is not blank-padded to this size (FIXED BIN(15)).

status — Return code (FIXED BIN(15).

Example

```
OK, SLIST CLGET1.PASCAL

{<readtty.pascal> Reads text from the user terminal using the external}
{                 PRIMOS routine CL$GET                                }
{                                                                      }
{Thisprogram provides an example on how define a suitable Pascal       }
{structure for implementing the character varying datatype found in    }
{PL1G. Since standard Pascal prohibits reading string data from files  }
{without subscripts, this example will provide an alternate            }
{solution for reading strings from the user terminal, without          }
{explicit     subscripting.                                            }
{                                                                      }
{ The simple object of the program is to read 3 strings from the       }
{ terminal and display them in complete reverse order.                 }
{                                                                      }
program readTTY;
type

  char80varying =
    record
      l : integer;
      s : array[1 .. 80] of char;
    end;
```

```
var
  cmdline : char80varying;
  table   : array[1 .. 3] of char80varying;
  i,j     : integer;
  status  : integer;

procedure cl$get(var cmdline: char80varying;{Command line input buffer}
                     lenBytes: integer;     {Length of cmdline in bytes}
                 var status   : integer);    {Return error code status }
          extern;                            {External PRIMOS procedure}

begin
  { Loop to input the text entered from the user terminal using the  }
  {  PRIMOS routine defined above (cl$get).                          }
  {                                                                  }
  for i := 1 to 3 do
  begin
    write(i:1,'> ');
    cl$get(cmdline, 80, status);
    if status <> 0
      then
        writeln('Bad status code returned, status =',status);
    table[i] := cmdline;  { save the command line }
  end;

  { Display the lines just typed in reverse order. }
  writeln;

  for i := 3 downto 1 do
  begin
    write(i:1,'< ');
    for j := table[i].1 downto 1 do
      write(table[i].s[j]);
    writeln;  end;
  end.
```

This program, stored as CLGET1.PASCAL, may be compiled, loaded, and run with the following dialog:

```
OK, PASCAL CLGET1
0000 ERRORS (PASCAL-REV 19.0)
OK, SEG -LOAD
[SEG rev 19.0]
$ LO CLGET1
$ LI PASLIB
$ LI
LOAD COMPLETE
$ EXEC
1> ABCDE
2> SECOND
3> MADAMIMADAM
```

```
3< MADAMIMADAM
2< DNOCES
1< EDCBA
OK,
```

▶ CL$PIX

Purpose

This subroutine parses command arguments according to a character string "picture" of the command line. It allows a program to process arguments on a command line, using the rules explained for arguments in Chapter 13 of the CPL User's Guide.

The caller supplies the command argument picture, the command arguments to parse, an output structure whose shape corresponds left-to-right with the picture, and other parameters. CL$PIX parses the picture and, if the picture is valid, parses the command arguments into the supplied structure. At that point, the individual arguments have been validated to be of the correct data type, converted if necessary, and are accessible to the program in a straightforward manner.

19

Usage

```
DCL CL$PIX ENTRY (BIT(16) ALIGNED, CHAR(*)VAR, PTR, FIXED BIN,
          CHAR(*)VAR, PTR, FIXED BIN, FIXED BIN, FIXED BIN, PTR);
```

```
CALL CL$PIX (keys, caller-name, picture-ptr, pixel-size,
             com-args, struc-ptr, pix-index, bad-index,
             code, local-vars-ptr)
```

keys — A 16-bit word that is input to control certain details of processing. The bits of keys have the following structure:

```
 ______________________________________________________
|  1|  2| ...                    | 13| 14| 15| 16|
 ------------------------------------------------------
```

The structure may be used in any language as a 16-bit integer with a value equivalent to setting on the bits desired. The PL1G data description for this structure is:

```
1 keys,
   2 debug bit(1)
   2 mbz bit(11), /* must be '0'b — 11 bits*/
   2 keep_quotes bit(1),
   2 cpl_flag bit(1),
   2 pl1_flag bit(1),
   2 no_print bit(1);
```

If no_print is '1'b, no error messages will be printed by CL$PIX; only error code information will be returned. If no_print is '0'b, caller-name is used to format the error message. (See below.)

If pl1_flag is '1'b, the Pl/I data type "bit(1) aligned" will be used for control_argument presence flags in the output structure. (See below.) If pl1_flag is '0'b, the FORTRAN data type LOGICAL (PL1G data type "bit(16) aligned") is used instead.

If cpl_flag is '1'b, CL$PIX operates in CPL mode; otherwise, it operates in normal mode. These modes are explained below. Most callers will want to use normal mode. 19

If keep_quotes is '1'b, CL$PIX will not strip quotes from parsed string arguments; otherwise, it will remove one layer of quotes. This flag is ignored in CPL mode, and quotes are never stripped.

If debug is '1'b, CL$PIX will print on the terminal a dump of the parsed argument picture. This is not useful for most applications programmers.

caller_name — The name of the calling routine (input). This name will be used to format error messages, if no_print is '0'b.

picture-ptr — A pointer to a varying character string containing the command argument picture (input). If dimensioned, the array must be connected (contiguous). The syntax and semantics of the picture are defined below.

pixel-size — The maximum length in characters of the element(s) of the object pointed to by picture-ptr (input). This provision allows an arbitrarily large array of strings to be passed and circumvents compiler restrictions on character-string length.

com-args A string containing the command arguments to be parsed (input). It is not necessary to translate this string to uppercase only, or do any other preprocessing on it. All syntactic conventions of the PRIMOS Command Language, including the "/*" comment delimiter, are supported.

struc-ptr A pointer to an output structure whose members will be filled in with the results of a valid picture parse of the supplied command arguments. (This argument is used only in normal mode; in CPL mode, local-vars-ptr determines the destination of the output of the parse.) The format of this structure is determined by the components of the picture, and is described below (input, addresses output).

pix-index This is valid only when code is nonzero (returned). When valid, pix-index is 0 if the error applies to the command arguments string, and is i if the error applies to element (pixel) i of the picture itself. Errors in the picture are fatal in the sense that no attempt is made to parse the command arguments if the picture cannot be parsed.

19 bad-index The character index (counting from 1) of the first character of the token (word or expression) causing the error (returned). The value of pix-index must be consulted to determine whether bad-index is relative to the command arguments or to a pixel of the picture. bad-index is valid only if code is nonzero.

code A nonstandard return code, which can take on the following values:

0 No error.

1 Null argument group (two successive semicolons) in picture.

2 Missing or illegal delimiter in picture.

3 Illegal option argument name in picture.

4 Illegal repeat count in picture.

5 Unknown data type name in picture.

6 Implementation error in picture parse.

7 A token was longer than 1024 characters in picture.

8 Option arguments precede object arguments in picture.

11 Too many object arguments in command line.

12 Option argument appears in command line that is not specified in the picture.

13 Object or parameter on command line does not have the correct format for its data type.

14 Default value not in proper format in picture.

15 Default value may not be given for this data type.

16 Too many instances of an option in command line.

17 A default value expression contains an undefined variable reference or a format error. (CPL mode only.) 19

18 The data type UNCL has been given more than once or has been given for an option argument parameter.

local-vars-ptr A pointer used only in CPL mode (input and return). In this case, it is a pointer to the Local Variable Control Block that identifies the local variable area to be used to hold the parsed arguments. local-vars-ptr should be null if not in CPL mode. See the description of CPL mode below.

The Picture in Normal Mode

This mode is used by most callers of CL$PIX. It is intended to be used by a command to process its command-level arguments into a form that it can use for decision making or further processing. It is a CHAR(*)VAR string, and must be scalar (singly-dimensioned).

Basic Format: The syntax of the normal mode picture is very similar to that of the CPL &ARGS directive, the major difference being that no variable names are allowed (because the results are not being stored in local command variables).

The picture looks like:

```
argument group [; argument group]; ...; end
```

Each argument group defines either an object argument, or an option argument and its associated objects if any. The end token is required to delimit the end of the picture string, and must be last in the string.

First, a word about lexical format. Upper- and lowercase are equivalent anywhere except inside quotes. Extra blanks may appear anywhere that a single blank is allowed or required. Blanks are not required to precede or follow other delimiters, such as ";", but they may be present if desired. Single character string tokens that contain blanks or delimiters must be enclosed in quotes, but the quotes are not part of the token itself. The delimiter characters are:

```
blank , ; = ( ) * %
```

Other punctuation or special characters should also be quoted.

If the picture is supplied in the form of an array of varying strings, an implicit lexical blank separates elements of the array. That is, when the end of any element is reached, a blank is recognized, regardless of the length of that particular element.

19

Object Argument Groups: As in the CPL &ARGS directive, all <argument groups> that define object arguments must appear before the first <argument group> that defines an option argument.

The simplest <argument group> simply declares the data type of the object argument. CL$PIX supports the following data types:

char — Arbitrary character string up to 80 bytes long, mapped to uppercase.

charl — Arbitrary character string up to 80 bytes long, not mapped.

tree — PRIMOS pathname up to 128 bytes long, mapped to uppercase. Wildcard characters are allowed.

entry — Filename, up to 32 bytes long, mapped to uppercase. Wildcard characters are allowed.

id — PRIMOS user or project identifier, up to 32 bytes long, mapped to uppercase. Must begin with a letter, and contain only letters, digits, or the special characters "$", ".", or "_".

password PRIMOS user login password, up to 16 bytes long, mapped to uppercase. May contain any characters except PRIMOS reserved characters.

dec Decimal integer with optional sign, in the range (2**31 - 1) to (-2**31 + 1).

oct Octal integer with optional sign, in the range (2**31 - 1) to (-2**31 + 1).

hex Hexadecimal integer, unsigned, in the range 0 to (2**32 - 1).

date A calendar date and time in one of the standard formats:

ISO (YY-MM-DD.HH:MM:SS.dow)

USA (MM/DD/YY.HH:MM:SS.dow)

Visual (DD Mmm YY HH:MM:SS day-of-week)

The day of week field is always ignored (but checked for legality); time fields default to 0; omitted YY defaults to current year; if entire date and "." are omitted, defaults to current date. The converted representation is the PRIMOS file system format. 19

ptr PRIMOS virtual address in the form S/W, where S is the octal segment number and W is the octal word number.

REST Rest of command line, up to 160 bytes long. (See below for explanation.) Upper- and lowercase are distinguished. See the discussion of data type REST below.

UNCL String of "unclaimed" tokens; that is, all tokens on the command line not accounted for elsewhere in the picture. Up to 160 bytes long. Upper and lower case are distinguished. See the discussion of data type UNCL below.

file Primos filename.

A simple picture might then be:

```
char; end
```

which defines a command line consisting of a single character string argument that will be mapped to uppercase. A more complex picture might be the following.

```
tree; dec; charl; end
```

This specifies three arguments: a treename, followed by a decimal integer, followed by a character string (unmapped).

Assignment to the Output Structure: When the command line is parsed against the picture, the structure pointed to by struc-ptr is filled in. The shape of the structure is determined by the picture: each object argument, option argument, or option argument parameter generates a member of the structure. The data type of each member is determined by the corresponding data type in the picture. The correspondence is:

19

Data Type	PL1G Type	FORTRAN Type
char	char(80) var	INTEGER(41)
charl	char(80) var	INTEGER(41)
tree	char(128) var	INTEGER(65)
entry	char(32) var	INTEGER(17)
id	char(32) var	INTEGER(17)
password	char(16) var	INTEGER(9)
dec	fixed bin(31)	INTEGER*4
oct	fixed bin(31)	INTEGER*4
hex	fixed bin(31)	INTEGER*4
date	fixed bin(31)	INTEGER*4
ptr	ptr options(short)	INTEGER*4
rest	char(160) var	INTEGER(81)
UNCL	char(160) var	INTEGER(81)
file	char(128) var	INTEGER(65)

Examples are:

```
Picture                  Structure

char;  end               dcl 1 struc,
                             2 char_arg char(80) var;

tree; dec; charl; end    dcl 1 struc,
                             2 tree_arg char(128) var,
                             2 dec_arg fixed(31),
                         2 charl_arg char(80) var;
```

Use of Data Types REST and UNCL: These two data types cause special processing to occur.

The UNCL data type can only be used with an object argument, not an option argument. Any token on the command line that does not match (is not "claimed" by) any part of the picture is added to the UNCL argument if one has been defined. A single blank separates each token added. If no UNCL argument is defined, unclaimed tokens are erroneous and the user's command line is in error. An example is shown under the option argument section, since with only object arguments in the picture and on the command line, the REST and UNCL arguments perform the same function. This is because scanning proceeds left to right, and all arguments on the command line that also appear in the picture must necessarily be claimed.

The REST data type can be used with either kind of argument; option arguments are explained below. When used with an object argument, if the REST argument is reached in the picture and more text remains on the command line, the entire remaining text is assigned to the REST argument. For example, in:

```
dec; tree; rest    (picture)

dcl 1 struc,       (structure)
     2 dec_arg fixed(31),
     2 tree_arg char(128) var,
     2 rest_arg char(160) var;
```

19

```
786 a>b>c>d foo 99 zot>nil     (command line)
```

786 is assigned to struc.dec_arg, a>b>c>d to struc.tree_arg, and foo 99 zot>nil to struc.rest_arg.

Default Values: What happens if an argument specified in the picture is not supplied by the user? In the absence of contrary instructions, the corresponding structure element is assigned a "default default" value, which is the null string for string types, 0 for arithmetic types, and null () for the pointer type.

The picture may specify some other default value. The syntax is:

```
data type = default-value;
```

For example:

```
tree = @.list;  dec = 99;  date = 81-1-1;  end

dcl 1 struc,
     2 tree_arg char(128) var,
     2 dec_arg fixed(31),
     2 date_arg fixed(31);

(null command line)
```

would assign @.LIST (note uppercase conversion) to struc.tree_arg, 99 to struc.dec_arg; and 81-01-01.00:00:00 (in file system format) to struc.date_arg.

Repeat Counts: To save typing, a repeat count feature is included in the syntax. To use it, simply prefix the <argument group> to be duplicated with the repeat count followed by "*". For example:

```
5 * dec = -1; 2 * char = foo; end

dcl 1 struc,
     2 dec_args(5) fixed(31),
     2 char_args(2) char(80) var;
```

The repeat count must be positive and less than 1000.

Note the use of arrays in the structure above. This is not required; one could employ five scalar fixed(31) members with different names in place of dec_args, for example.

Option Arguments: CL$PIX allows convenient handling of PRIMOS command line option arguments. An <argument group> that specifies an option argument is distinguished from an object argument group by beginning with a "-". The general form is: 19

```
-name1, -name2, ..., -namen {<obj1> <obj2> ...};
```

The -names are the names of the option argument as the user will use them on the command line. Multiple names are allowed to enable the definition of synonyms and abbreviations.

The simplest option argument has no parameters. An example is:

```
-listing, -l

dcl 1 struc,
     2 listing_arg bit(1) aligned;
```

Note

The data type used for all option arguments is controlled by a flag in the keys argument to CL$PIX. (See above.) Here, assume that keys.pl1_flag is '1'b.

The struc.listing_arg will be set to '1'b if -LISTING or -L appears on the command line; otherwise it is set to '0'b. There is no default value for a simple option argument: it either is or is not on the command line. Hence the "=" syntax is not relevant here.

If an option argument is to have parameters, they are the objs in the general form, and are specified using the syntax for object <argument group>s. Suppose that option -LISTING is to accept a treename parameter. The following could be used:

```
    -listing, -l tree = listing.list;  end

    dcl 1 struc,
         2 listing bit(1) aligned,
         2 listing_tree char(128) var;
```

If a treename follows -LISTING on the command line, it is assigned to struc.listing_tree; otherwise struc.listing_tree is assigned LISTING.LIST. Note that the default values are assigned to parameters of an option even if that option is not given on the command line.

As another example, an option -RANGE is to take two integer parameters:

```
    -range dec = 0;  dec = 99999;  end

    dcl 1 struc,
         2 range_bit(1) aligned,
         2 range_lower fixed(31),
         2 range_upper fixed(31);

    -range 7        (command line)
```

19

struc.range is '1'b, struc.range_lower is 7, and struc.range_upper is 99999 (the default).

Using the REST Data Type with Option Arguments: The REST data type can be used as the data type of the rightmost parameter of an option argument. For example:

```
    char;  -string rest;  -page dec = 1;  end

    dcl 1 struc,
         2 char_arg char(80) var,
         2 string_flag bit(1) aligned,
         2 string_rest char(160) var,
         2 page_flag bit(1) aligned,
         2 page_number fixed(31);
```

When the option -STRING is seen on the command line, the entire remainder of the command is assigned to the REST argument, in this case struc.string_rest. For example:

```
    foo -page 17 -string abc def -page 0
```

assigns 'FOO' to struc.char_arg, '1'b to struc.string_flag, 'abc def -page 0' to struc.string_rest, '1'b to struc.page_flag, and 17 to struc.page_number.

Note that CL$PIX (at least) is not confused by the second occurrence of -page: it is part of struc.string_rest because it follows the -string option.

Using the UNCL Data Type with Option Arguments: The data type UNCL may only be assigned to an object argument, not to the parameter of an option argument. However, it is possible for option arguments to be unclaimed and hence added to the UNCL argument.

Consider the problem: write a command interface that accepts a treename object argument and the option argument -time with an integer parameter, but which accepts and passes on all other arguments to some other interface.

A picture to do this is:

```
tree; UNCL; -time dec; end
```

```
dcl 1 struc,
     2 tree_arg char(128) var,
     2 UNCL_arg char(160) var,
     2 time_flag bit(1) aligned,
     2 time_number fixed(31);
```

19 Then the command:

```
a>b>c zot -lines 78 -time 88 def -zilch a b c
```

sets struc.tree_arg to 'A>B>C', struc.UNCL_arg to 'zot -lines 78 def -zilch a b c', struc.time_flag to '1'b, and struc.time_number to 88. Note particularly that def is not a parameter of -time but an object argument. Since the TREE argument was already accounted for, def was unclaimed. the command:

```
-limits abc def -time 90 a>b>c
```

sets struc.tree_arg to 'A>B>C', struc.UNCL_arg to '-limits abc def', struc.time_flag to '1'b, and struc.time_number to 90.

Note

Why did struc.tree_arg not get assigned the value 'ABC' or 'def'? Because of the rule given for UNCL above:

All parameters that follow an unclaimed option argument will be considered unclaimed. This is because the picture contains no information about an unclaimed option argument, and hence CL$PIX cannot know how many parameters may follow it.

Thus all object arguments following an unclaimed option argument are taken as parameters of that option, until a claimed option argument is found.

Multiple Instances of an Option Argument: A picture may contain more than one instance of the same option argument. It is recommended that each instance contains exactly the same synonym or abbreviation names for the option, though CL$PIX does not check for this.

When multiple instances are used, the semantics are that multiple instances of the option on the command line are permitted, and will appear in successive slots of the output structure. The usual use of this capability is best illustrated by an example.

Suppose that a command accepts an option -select with one parameter, say a string to search for in a file. It seems reasonable to allow the command to search for multiple strings at once; hence the desire for multiple instances of the option. A picture might be:

```
    -select charl;  -select charl;  -select charl;  end
```

which allows for three instances of -select. The structure is:

```
    dcl 1 struc,                                                    19
         2 select_1 bit(1) aligned,
         2 select_1-char char(80) var,
         2 select_2 bit(1) aligned,
         2 select_2-char char(80) var,
         2 select_3 bit(1) aligned,
         2 select_3-char char(80) var;
```

The first -select encountered goes into struc.select_1, the second into struc.select_2, and the third into struc.select_3. Note that the three instances need not follow each other directly in the picture; and, if they do not, they will not follow each other in the structure. Thus the existence of multiple instances of an option does not alter the usual left-to-right assignment of argument groups to structure member slots.

Any option argument that appears only once in the picture may appear at most once on the command line.

Using Repeat Counts with Option Arguments: Repeat counts can be used with option arguments in a fashion analogous to their use with object arguments. They are simply a typing saver. Consider the "-select" example above. An equivalent picture is:

```
    3 * -select charl;  end
```

That is, a repeat count used in this way declares multiple instances of an option argument, together with its parameters. It is also possible to use repeat counts on the parameters. Consider the following picture:

```
3 * -limits 2 * dec = 0;  end
```

It is the same as:

```
-limits dec = 0 dec = 0;  -limits dec = 0 dec = 0;
    -limits dec = 0 dec = 0;  end
```

The Picture in CPL Mode

Syntax Differences: The syntax of the picture accepted in CPL mode is exactly the same as that accepted by the CPL &ARGS directive. (In fact, CPL uses CL$PIX in CPL mode to process the &ARGS directive.) See the CPL User's Guide for full details.

The salient differences between normal and CPL mode syntaxes are:

- Repeat counts are not allowed in CPL mode.

19

- Each object argument and option argument must be preceded with the syntax:

  ```
  <variable-name>:
  ```

  where <variable-name> is a legal CPL local variable name. The value of each argument will be assigned to the local variable whose name is prefixed to that argument.

- The maximum size of any argument value in CPL mode is 1024 characters, unlike normal mode where the limit depends on the data type (80 characters for CHAR and CHARL, 160 for REST, and so on).

Local Variable Storage Management: In CPL mode, it is quite possible for CL$PIX to run out of room in the supplied Local Variables Area while attempting to set the values of all the local variables involved. If this happens, CL$PIX will return the error code E$ROOM.

It is the caller's responsibility at this point to allocate more space for the Local Variables Area, and to call CL$PIX to redo the parse from the start. This process may have to be repeated in a loop until enough storage has been added to accommodate the values of all the local variables involved.

Usage Differences: In CPL mode, the "end" keyword is not required to appear at the end of the picture. For this reason, a picture array is not allowed: the picture must be supplied as a one-dimensional (scalar) varying string up to 1024 characters long.

19

Calls Made by CL$PIX

TNCHK$, FNCHK$, IDCHK$, PWCHK$.

▶ CNIN$

Purpose

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.

Usage

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

buffer — A buffer in which the string of characters read from the input stream is to be placed, two characters per word (integer array).

char-count — The number of characters to be transferred from the input stream to buffer (INTEGER*2).

actual-count — A returned argument (INTEGER*2). It specifies the number of characters read by the call to CNIN$. If reading continues until a NEWLINE character is encountered, the count includes the NEWLINE character.

Discussion

CNIN$ reads from the input stream until either a NEWLINE character is encountered or the number of characters specified by char-count is read. Characters are left-justified, and if an odd number of characters is read, the remaining character space is not zero- or blank-filled. The line-delete and character-delete 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 Prime User's Guide for further information.)

▶ COMANL

Note

18.1 For PL1G and Pascal programmers, this subroutine is obsolete and has been replaced by CL$GET.

Purpose

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.

Usage

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$

Purpose

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

Usage

CALL DUPLX$ (tcw)

int*2 = DUPLX$(tcw)

tcw Terminal configuration word: a 16-bit integer whose bits have the following meanings (input and output):

Bit	Mask	Meaning if Bit is SET
1	100000	Half duplex.
2	040000	Do not echo LINEFEED after CARRIAGE RETURN.
3	020000	Turn on XOFF/XON character recognition.
4	010000	Output currently suppressed (XOFF received).
5		Detect DATA SET BUSY before output to AMLC line. (See AMLC Functions below.)
6		Handle reverse channel functionality. (See AMLC Functions below.)

Data Set	Sense Bits	
(INA '0054)	Bit 6=1	Bit 6=0
1 (off)	XOFF	XON
0 (on)	XON	XOFF

7		Check for certain error conditions: ● Overflow of the input buffer

		• Parity error If one of these conditions is present, the character found is replaced with '225.
8		Indicates a parity error (output). Overflow of the input buffer is flagged when there is only room for one more character.
9-16	000377	Internal buffer number (read-only).

Discussion

DUPLX$ has no effect under PRIMOS II.

DUPLX$ returns the terminal configuration word and internal buffer number as the value of the function. DUPLX$ must be declared as a 16-bit INTEGER function if the returned value is to be used by the calling program.

If the terminal configuration word passed to DUPLX$ is equal to -1, no updating of the configuration word takes place. In this case, the current value is returned.

The tcw input from a user terminal is not affected by the LOGIN or LOGOUT commands. The tcw of the user terminal may also be set at the supervisor terminal by using the AMLC command. Users may also use the PRIMOS command TERM to change their terminal characteristics.

AMLC Functions

Certain devices require a reverse channel protocol to signal BUSY or READY. For these cases, the carrier detect line is used for the signal. Bit 5 of the terminal configuration word will instruct the AMLDIM to interrogate the carrier signal for that line before outputting. If a BUSY is detected, then the AMLDIM will simulate an XOFF received for that ine. When the carrier signal goes to the READY state, the AMLDIM will flag it as an XON, and output will resume. For example, if the device signals BUSY as DATA SET off (1), then the terminal configuration word bit setting would be:

Bit 5 = 1 (detect DATA SET sense)

Bit 6 = 1 (if DATA SET sense is off, then simulate XOFF, else set XON.)

▶ ERKL$$

Purpose

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

Usage

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

key — An INTEGER*2 specifying the action to be taken. Possible values are:

K$WRIT — Set erase and kill characters.

K$READ — Read erase and kill characters.

erase — With key K$WRIT, the character contained in the right byte of erase replaces the user's erase character. If erase is 0, no action takes place. On key K$READ, the user's current character is placed in erase, right-justified with leading zeros.

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

code — An INTEGER*2 variable set to the return code. Possible values are:

0 — No errors.

E$BPAR — Attempt to set characters is improper.

Discussion

Erase and kill characters are interpreted by commands to the operating system and through the subroutines COMANL, RDTK$$, RDASC, I$AA12, and I$AA01. All language processors and I/O statements call RDASC to get terminal input and, therefore, are affected.

Note: RDASC, I$AA12, and I$AA01 are library subroutines that read the user's erase and kill characters 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. Thus, 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. The characters are reset to default values upon an explicit logout or login.

▶ ERRPR$

Purpose

ERRPR$ interprets a return code and, if it is nonzero, prints a standard message associated with the code, followed by optional user text. See Appendix D for more details on error handling.

Usage

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

key — An INTEGER*2 specifying the action to take after 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*2 variable containing the return code from the routine that generated the error. If code is 0, 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 0 (integer array).

txtlen The length in characters of text (INTEGER*2).

filnam The name of the program or subsystem detecting or reporting the error. filnam is omitted by specifying both filnam and namlen as 0 (integer array).

namlen The length in characters of filnam (INTEGER*2).

Discussion

More explanation of the use of ERRPR$ is given in Appendix D.

▶ EXIT

Purpose

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

Usage

```
CALL EXIT
```

▶ FNCHK$

Purpose

This function checks the name passed for validity as a filename. This means that the name may not contain PRIMOS reserved characters, lowercase letters, or control characters, may not start with a digit, and must be between 1 and 32 characters long. The keys passed to FNCHK$ may modify these restrictions. 19

Usage

```
DCL FNCHK$ ENTRY (FIXED BIN, CHAR(*)VAR) RETURNS (BIT(1));

name-ok = FNCHK$ (key, filename);
```

key — Defines restrictions on filename. Keys may be added together:

- K$UPRC — Mask name to uppercase before checking.
- K$WLDC — Allow wildcards in name.
- K$NULL — Allow null names.
- K$NUM — Allow numeric names (segment directory entry names).

filename — Name to be checked (input only unless K$UPRC is used; in that case, input/output).

name-ok — Set to PL1G true if the name is valid given the restrictions of the keys.

18.1

▶ GCHAR

Purpose

GCHAR gets a character from an array. This subroutine is helpful, for example, in retrieving character information for a FORTRAN program.

Usage

```
char = GCHAR (LOC(array), index)
```

array — Array of characters.

index — Index of the location of character in array (INT*2).

Discussion

The pointer (index) must be initialized by the user to 0 and is incremented by 1 after the operation is complete.

▶ GINFO

Purpose

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

Usage

CALL GINFO (xervec, n)

GINFO returns 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	Reserved
4	Reserved
5	Low boundary of PRIMOS II and buffer (64K for PRIMOS II only)
6	High boundary of 64K PRIMOS II

Information for PRIMOS:

xervec Word	Content
1	0
2	0
3-6	Reserved

▶ GV$GET

Purpose

GV$GET retrieves the value of a global variable.

Usage

```
DCL GV$GET ENTRY (CHAR(*)VAR, CHAR(*)VAR, FIXED BIN, FIXED BIN)

CALL GV$GET (var-name, var-value, value-size, code)
```

var-name — The name of the global variable whose value is to be retrieved. The name must follow the rules for CPL global variable names and must be in uppercase. It must be in the global variable file last invoked with DEFINE_GVAR.

18.1 var-value — The returned value of variable var-name.

value-size — The length of the user's buffer var-value in characters.

code — A return code:

E$BFTS	The user buffer var-value is too small to hold the current value of the variable.
E$UNOP	The global variable storage is uninitialized or in bad format.
E$FNTF	The variable is not found.

Discussion

The PRIMOS command DEFINE_GVAR must be used to define the global variable file before this subroutine is called. For more information on global variables, see the CPL User's Guide.

▶ GV$SET

Purpose

GV$SET sets the value of a global variable.

Usage

```
DCL GV$SET ENTRY (CHAR(*)VAR, CHAR(*)VAR, FIXED BIN)

CALL GV$SET (var-name, var-value, code)
```

var-name — The name of the global variable to be set. This name must follow the rules for CPL global variable names. All letters must be uppercase.

var-value — The new value of the variable <u>var-name</u>.

code — A return error code:

- E$BFTS — The specified value is too big.
- E$UNOP — The global variable area is bad or uninitialized. 18.1
- E$ROOM — An attempt by the variable management routines to acquire more storage fails.

Discussion

The PRIMOS command DEFINE_GVAR must be used to define the global variable file before this subroutine is called. For more information on global variables, see the <u>CPL User's Guide</u>.

▶ IDCHK$

Purpose

19

This function checks that the name passed is a legal user or project id. This means that the name must be between 1 and 32 characters long, start with an uppercase letter, and contain only uppercase letters, numbers, and the special characters . $ and _ .

Usage

```
DCL IDCHK$ ENTRY(FIXED BIN, CHAR(*)VAR) RETURNS (BIT (1));

id-ok = IDCHK$(key, id);
```

key — Restrictions on the name. Keys may be added together:

K$UPRC — Mask id to uppercase before checking. 19

K$WLDC — Allow wildcard characters in the id. (See the Prime User's Guide.)

K$NULL — Allow null id's.

id — The id to check (Input unless key is K$UPRC; in that case, input/output.)

id-ok — Set to PL1G true if the name is valid given the restrictions of the keys.

▶ LOGO$$

Purpose

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.

Usage

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

key — Operation to be performed (INTEGER*2). Possible values are the following.

-1 — Log out all users (supervisor only).

0 — Log out self (same as LOGOUT command).

1 — Log out specific user by number (same as LOGOUT -NN).

2 — Log out specific user by name (supervisor or its phantoms only).

user	User number to be logged out. This value is examined only if key > 0 (INTEGER*2).
usrnam	Name of user to be logged out; must correspond to number supplied in user. This value is examined only if key is 2 (integer array).
unlen	Length of usrnam in characters. This value is examined only if key is 2 (INTEGER*2).
reserv	Reserved for future use (INTEGER*4).
code	Return code (INTEGER*2). Possible values are:

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

▶ LON$CN

Purpose

This PL1G subroutine is used to turn off or turn on logout notification. When notification is turned off, phantom logout information is queued (first-in first-out). When notification is turned on, queuing is not performed, and the default on-condition, PH_LOGO$, is raised if there is any logout notification data to be received.

See the discussion of LON$R for more information.

18.1

Usage

```
CALL LON$CN (key);
```

key	Software interrupt status key (FIXED BIN(15));

0	Notify off.
1	Notify on.

▶ LON$R

This PL1G subroutine fetches or transfers logout information from storage to a designated target area. It will do this unless it finds no information to transfer. The target area is designated by the argument msgptr. The size of the area pointed to by msgptr is designated by the argument msglen. The area should be at least six words in length. If it is shorter than this, LON$R will only fetch as much information as msglen can hold.

LON$R also passes back to its caller an indication whether there have been more phantom logouts with their information stored in a queue. This indication is contained within the argument more.

An error code is returned to the user via the argument code.

Usage

CALL LON$R (msgptr, msglen, more, code); 18.1

msgptr — Area of memory in which to store message (POINTER type). Its format is shown below.

msglen — Length of area in which to store message (FIXED BIN(15)).

more — BIT(1):

0 — Indicates no more messages left on queue.

1 — Indicates more messages left on queue.

code — Return code (FIXED BIN(15)):

E$NDAT — No data found in queue.

E$BFTS — Some information lost during transfer (msglen less than actual message length).

MSGPTR Area Format

Word Number	Information
1	Phantom's user number (fixed bin(15))
2	Time of day of logout (fixed bin(15))
3	Connect time in minutes (fixed bin(15))
4	CPU time in seconds (fixed bin(15))
5	I/O time in seconds (fixed bin(15))
6	Logout condition code (fixed bin(15)):
	0 Normal logout
	1 Abnormal logout

Discussion

A phantom is a process that can operate separately from its creator 18.1
process, and can continue working after the user has logged out. Phantoms are discussed in detail in the Prime User's Guide.

Logout Notification for Phantoms

Logout notification provides the creator of a phantom process with information about the phantom's activities. This information is compiled at phantom logout time and sent to the creator. This is known as notification.

Normally, the information will be displayed upon the creator's terminal. The information contains the phantom's user number, the time of day of logout, the elapsed time, CPU time, I/O time spent by the phantom, and an error code indicating normal or abnormal logout. Normal logout occurs when a phantom completes with a LOGOUT command. All other logout will generate abnormal logout.

Logout information will be compiled at this time and sent to the creator. If a user is logged into more than one terminal, the information will only be sent to the terminal from which the phantom was created. If the creator of the phantom has logged out while the phantom was running, the information will not be sent. This means that once a user has logged out, the phantom will not notify the user of logout even if the user logs back in.

Sometimes it may become necessary for a user to record the phantom logout information via a program. The logout notification system provides two subroutines that allow for this event. The first subroutine, LON$CN, allows a user to turn logout notification off or on. The second subroutine, LON$R, allows a user to fetch phantom logout information instead of having the information written to a terminal.

When LON$CN is requested to turn off logout notification, phantom logout information is automatically queued for future access. This allows users to turn off notification without having to worry about either the condition of their terminal screen or the loss of the status of their phantoms.

When LON$CN is requested to turn on logout notification, any enqueued
18.1 logout information is written on the user's terminal.

As mentioned above, a user may fetch phantom logout information by invoking LON$R. Normally, logout notification is enabled and invoking LON$R will gain the user nothing. Therefore, when using LON$R, logout notification should be turned off by invoking LON$CN.

When logout notification occurs, a system default condition handler or on-unit named PH_LOGO$ is invoked to write the information upon the creator's terminal. This on-unit is also invoked when LON$CN is requested to turn on logout notification. Therefore, users who do not ever wish to see logout information written upon their terminal should create their own on-unit and name it PH_LOGO$. This user-defined PH_LOGO$ should usually call LON$R to fetch phantom logout information, since the default PH_LOGO$ does this. On-units are discussed in Chapter 22.

▶ PHANT$

Note

19 This subroutine may be used only for non-CPL phantoms. It has been replaced with PHNTM$.

Purpose

PHANT$ starts a phantom user.

Usage

```
CALL PHANT$ (filnam, namlen, funit, user, code)
```

filnam — Name of command input file to be run by the phantom (integer array).

namlen — Length of characters of filnam (16-bit integer).

funit — File unit on which to open filnam. If funit is 0, unit 6 will be used (16-bit integer).

user — A variable returned as the user number of the phantom (16-bit integer).

code — The return code (16-bit integer). If it is 0, the phantom was initiated successfully. If code is E$NPHA, no phantoms were available. Other values of code are file system error indications.

▶ PHNIM$

Purpose

This subroutine allows a process to start up a phantom using either a command input file or a CPL file. See LON$R for a discussion of phantoms.

Usage

```
DCL PHNIM$ ENTRY (BIT(16) ALIGNED, CHAR(32), FIXED BIN, FIXED BIN,
                  FIXED BIN, FIXED BIN, CHAR(128), FIXED BIN)

CALL PHNIM$ (cplflg, filename, name-LEN, funit, phant-user,
                  CODE, ARGS, ARGSL)
```

19

cplflg — Source of process: if true ('1'b), then a CPL program is being started as a phantom; if false ('0'b), then a cominput file is being started as a phantom. (BIT(16) aligned = LOGICAL.)

filename — The name of the file to be started as a phantom.

name-len — The number of characters in filename.

funit — The file unit on which to open the phantom file.

user — The user number of the phantom (returned).

code — A return code; 0 means no error.

args — The arguments for a CPL phantom; a dummy argument must be given for non-CPL phantoms.

argsl — The number of characters in args; a dummy argument must be given for non-CPL phantoms.

Discussion

A phantom is a process that can operate separately from its creator process, and can continue working after the creator has logged out. Phantoms are discussed in detail in the *Prime User's Guide*. See LON$R for a discussion of phantoms also.

▶ PWCHK$

Purpose

19 This function makes sure that the password supplied is a legal login password.

Usage

```
DCL PWCHK$ ENTRY (FIXED BIN, CHAR(*)VAR) RETURNS (BIT(1));

pw-ok = PWCHK$(key, password);
```

key — An INTEGER*2 user option to restrict values of *password*. Keys may be added together:

K$UPRC — Change password to uppercase before checking.

K$NULL — Allow null passwords.

password — Must be 1 to 16 characters long, and may not contain lowercase letters or PRIMOS reserved characters.

pw-ok — Set to PL1G true if the password is legal.

▶ RDTK$$

Note

For PL1G and Pascal programmers, RDTK$$ is obsolete and has been replaced with CL$PIX above. 19

Purpose

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 token by token. A command line consists of tokens (defined below) 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(so to speak)'. 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 to subroutines that call these, such as FORTRAN formatted READ statements to the terminal, before parsing the command line. These subroutines cause the replacement of the information in the buffer holding the command line.

Usage

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

key — The action to be taken by RDTK$$ (INTEGER*2). 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 Initialize the command line.

info An eight-word integer array 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 are returned.)

2 Register setting parameter.

5 Null token.

6 End of line.

info(2) The length in characters of the token. A null token has a 0 length.

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

bit 1 (:100000) - Decimal conversion successful (no overflow), value returned in info(4).

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

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

bit 4 (:010000) - An explicit position for a register setting was given; position value is returned in info(4).

bits 5-16 Reserved.

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. Otherwise info(4) is undefined.

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

info(6)-(8) Reserved.

buffer An integer 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 (INTGER*2). buflen must be >= 0.

code A standard return code (INTEGER*2). 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 is 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 are discussed in the next paragraphs.

Blank Interpretation (SP): A single blank terminates a token. A multiblank 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 only 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 COBOL's or 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 0 to 80 characters in length. The following are examples of valid tokens:

```
FTN
LONG-FILENAME
1/707
6
98
String.even.longer.than.thirty-two.characters
[path]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
A0001
This.is.a.token.
PARTIALLY' L I T E R A L'
'8'xxx  (Note: '8' is treated as a nonnumeric.)
''''''''      (= ''')
```

Register-setting Token: Register-setting tokens (explained in the LOAD and SEG Guide) 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, for example, FTN XX -64V instead of FTN XX 2/400.

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 reenabled 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.

Strategy

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 was 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 0 can be used to skip over a token. The error code E$BFTS will be returned.

For a key of 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 uppercase is performed.

▶ RECYCL

Purpose

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

Usage

CALL RECYCL

Caution

Do not use this subroutine to simulate a time delay.

▶ SCHAR

Purpose

This subroutine stores a character into an array location. It is useful, for example, in storing character data from a FORTRAN program.

Usage 18.1

CALL SCHAR (LOC(array), index, char)

array	Array of characters
index	Index of the location of character in array (INT*2)
char	Character to be stored (one word)

Discussion

18.1 The pointer (index) is initialized to 0 and is incremented by 1 after the operation is complete.

The right half of the character word is used for storage, so for storing one character, the form of char should be ' A', for example.

▶ TEXTO$

Note

19 For PL1G and Pascal programmers, this subroutine is obsolete and has been replaced with FNCHK$.

Purpose

TEXTO$ checks a filename for valid format.

Usage

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

filnam — An integer array containing the filename to be checked.

namlen — The length of filnam in characters (INTEGER*2).

trulen — An (INTEGER*2) set to the true number of characters in filnam. trulen is valid only if textok is .TRUE..

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.

textok — A LOGICAL variable set to .TRUE. if filnam is a valid filename, otherwise set to .FALSE..

Caution

Names longer than 32 characters are truncated with no warning message.

Example

To read a name from the terminal, check for validity, and set trulen to the actual name length:

```
    CALL I$AA12 (0, BUFFER, 80, $999)
    CALL TEXTO$ (BUFFER, 32, TRULEN, OK) /* SET TRULEN
    IF (.NOT. OK) GOTO <bad-name>
```

▶ TIMDAT

Purpose

TIMDAT returns the date, time, CPU time, and disk I/O time used since login, the user's unique number on the system, and the user id in an array.

Usage

CALL TIMDAT (array, num)

19

array   An integer array:

1   Two ASCII characters representing month.

2   Two ASCII characters representing day.

3   Two ASCII characters representing year.

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.

10   Integer disk I/O time used in ticks.

11   Integer number of ticks per second.

12 User number.

13-28 Login name, left-justified.

19 num Must be 28 (INTEGER*2).

Discussion

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).

Examples

Use of TIMDAT is illustrated in sample programs in Chapters 3 through 8.

▶ TNCHK$

Purpose

This function checks the name passed for validity as a pathname.

Usage

```
DCL TNCHK$ ENTRY (FIXED BIN, CHAR(*)VAR) RETURNS (BIT(1));

name-ok = TNCHK$ (key, pathname);
```

19 key Determines the restrictions to be placed on the name. Keys may be added together:

K$UPRC Change name to uppercase before checking.

K$WLDC Allow wildcard characters in name. (See the Prime User's Guide.)

K$NULL Allow a null pathname.

pathname Must follow the rules for pathnames in Chapter 9 of this guide or in the Prime User's Guide, modified by the key above.

name-ok Set to PL1G true if the name is valid given the restrictions of the keys.

19

Discussion

Legal pathnames are discussed in Chapter 9. Filenames within the pathname are checked by FNCHK$, described earlier.

PART IV

MATH, SORT, and Applications Library Subroutines

11 FORTRAN Matrix Library (MATHLB)

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 11-1 for a summary.

These subroutines are available in R-mode only, so they may only be called from FORTRAN IV and PMA.

SUBROUTINE CONVENTIONS

The following conventions are used in the subroutine descriptions in this chapter.

Names

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

Table 11-1
Summary of Available Matrix Operations

Operation		Integer	Single Precision	Complex	Double Precision
Setting matrix to identity 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	CMSUB	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

Arguments

All arguments must be specified. Variables and arrays are assumed to be of the same mode as the subroutine (REAL, DOUBLE PRECISION, INTEGER*2, or COMPLEX). Matrix sizes and error flags must be declared as INTEGER*2.

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.

SUBROUTINE DESCRIPTIONS

▶ COMB

Purpose

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 reinitialized by the user by passing a restrt value of 1 along with new values for n and nr. (The restrt parameter is optional; if reinitialization is not desired, either omit this parameter from the calling sequence or set it to a value of 0).

Usage

```
CALL COMB (icomb, n, nr, iw1, iw2, iw3, last, restrt)
```

	Mode	Subscript(s)	Dimension(s)	Comments
icomb	Integer	1	nr	Return
n	Integer			Pass
nr	Integer			Pass
iw1	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, iw1, iw2, or iw3. For further details, see Gideon Ehrlich, "Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," *Journal of the ACM*, vol. 20, no. 3, July 1973, pp. 500-513.

▶ LINEQ

Purpose

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 nx1 column vector of unknowns in which the solution is stored.

Note

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

Usage

$$\text{CALL} \left\{\begin{matrix}\text{CLINEQ}\\ \text{LINEQ}\\ \text{DLINEQ}\end{matrix}\right\} \text{(xvect, yvect, cmat, work, n, npl, ierr)}$$

	Mode	Subscript(s)	Dimension(s)	Comments
xvect	*	1	n	Returned
yvect	*	1	n	Passed
cmat	*	2	n,n	Passed
work	*	2	np1,np1	Work
n	Integer			Passed
np1	Integer			Passed (=n+1)
ierr	Integer			Returned

* All of the same mode which determine the subroutine used

Discussion

The user is required to provide as a work area a np1xnp1 matrix (np1 = n+1). The integer error flag ierr returns one of three possible values:

ierr	Meaning
0	Solution found satisfactorily
1	Coefficient matrix singular
2	np1 < > n+1

If ierr < > 0, no modifications are made to xvect.

▶ MADD

Purpose

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.

Usage

$$\text{CALL}\left\{\begin{matrix}\text{DMADD}\\\text{CMADD}\\\text{IMADD}\\\text{MADD}\end{matrix}\right\}\text{(mats, mat1, mat2, n, m)}$$

	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

Purpose

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.

Usage

$$\text{CALL}\left\{\begin{matrix}\text{MADJ}\\\text{IMADJ}\\\text{CMADJ}\\\text{DMADJ}\end{matrix}\right\}\text{(mato, mati, n, iw1, iw2, iw3, iw4, ierr)}$$

	Mode	Subscript(s)	Dimension(s)	Comments
mato	*	2	n,n	Returned
mati	*	2	n,n	Passed
n	Integer			Passed
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

Discussion

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

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

Note

mato and mati must be distinct.

▶ MCOF

Purpose

Calculates the signed cofactor of the element mat (i,j) of the nxn matrix mat and stores this value in COF. If i = 0 and j = 0 the determinant of mat is calculated.

Note

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

Usage

CALL $\left\{\begin{matrix}\text{IMCOF}\\\text{CMCOF}\\\text{MCOF}\\\text{DMCOF}\end{matrix}\right\}$ (cof, mat, n, iw1, iw2, iw3, iw4, i, j, ierr)

	Mode	Subscript(s)	Dimension(s)	Comments
cof	*			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
i	Integer			Passed
j	Integer			Passed
ierr	Integer			Returned

* All of the same mode which determines the subroutine used

Discussion

The integer error flag ierr has two possible values:

ierr	Meaning
0	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 = 0 and j < > 0 or i < > 0 and j = 0 - subscript error 4. i>n and/or j>n - subscript error

▶ MCON

Purpose

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.

Usage

```
      ( IMCON )
      ) MCON  (
CALL  ) CMCON (  (mat, n, m, con)
      ( DMCON )
```

	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

Purpose

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.

Usage

$$\text{CALL}\left\{\begin{array}{l}\text{IMDET}\\ \text{MDET}\\ \text{CMDET}\\ \text{DMDET}\end{array}\right\}(\text{det, mat, n, iw1, iw2, iw3, iw4, ierr})$$

	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

Discussion

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

ierr	Meaning
0	Determinant formed successfully
1	n = 0 - no determinant possible

► MIDN

Purpose

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

```
MAT (I,J) = 0, I < > J
          = 1, I = J
```

Note

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

Usage

$$\text{CALL}\left\{\begin{array}{l}\text{IMIDN}\\ \text{MIDN}\\ \text{CMIDN}\\ \text{DMIDN}\end{array}\right\}(\text{mat, n})$$

	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.,0) (each SP)
Double-precision	DMIDN	1. (DP)

▶ MINV

Purpose

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 area, 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.

Usage

$$\text{CALL}\left\{\begin{matrix}\text{CMINV}\\ \text{MINV}\\ \text{DMINV}\end{matrix}\right\}\text{(mato, mati, n, work, npl, npn, ierr)}$$

	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 the subroutine used

Discussion

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

ierr	Meaning
0	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

Purpose

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.

Usage

$$\text{CALL} \left\{ \begin{matrix} \text{IMMLT} \\ \text{MMLT} \\ \text{CMMLT} \\ \text{DMMLT} \end{matrix} \right\} (\text{matp, matl, matr, n1, n2, n3})$$

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 the subroutine used

▶ MSCL

Purpose

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, respectively.

Usage

$$\text{CALL} \left\{\begin{array}{l}\text{IMSCL}\\ \text{MSCL}\\ \text{CMSCL}\\ \text{DMSCL}\end{array}\right\} \text{(mato, mati, n, m, scon)}$$

	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 the subroutine used

▶ MSUB

Purpose

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.

Usage

$$\text{CALL}\left\{\begin{array}{l}\text{IMSUB}\\ \text{MSUB}\\ \text{CMSUB}\\ \text{DMSUB}\end{array}\right\}\text{(matd, mat1, mat2, n, m)}$$

	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 used

▶ MTRN

Purpose

Calculates the transpose of the nxn matrix mati and stores it in the nxn matrix mato. The relationship between mati and mato is as follows: 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, CMTRN, or DMTRN, respectively.

Usage

$$\text{CALL}\left\{\begin{array}{l}\text{IMTRN}\\ \text{MTRN}\\ \text{CMTRN}\\ \text{DMTRN}\end{array}\right\}\text{(mato, mati, n)}$$

	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

Purpose

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 reinitialized 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 reinitialization is not desired either omit this parameter from the calling sequence or set it to a value of 0.) The calling program should not attempt to modify iperm, iw1, iw2, or iw3.

Usage

CALL PERM (iperm, n, iw1, iw2, iw3, last, restrt)

	Mode	Subscript(s)	Dimension(s)	Comments
iperm	Integer	1	n	Returned
n	Integer			Passed
iw1	Integer	1	n	Work

iw2	Integer	1	n	Work
iw3	Integer	1	n	Work
last	Integer			Returned
restrt	Integer			Passed (optional)

Discussion

For further details, see Gideon Ehrlich, "Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," Journal of the ACM, vol. 20, no. 3, July 1973, pp. 500-513.

12
Applications Library

GENERAL DESCRIPTION

This is a user-oriented library that provides a set of service routines, designed for ease of use. In many cases, the APPLIB or VAPPLB 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 or VAPPLB routines avoids a duplication of effort and provides a consistent interface for the terminal user.

All of these routines are written as FORTRAN functions that return one of the following: a status indication (logical .TRUE. or .FALSE.), an appropriate value, an alternate value or format of a returned argument, or a code 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.

These routines may be used either as subroutines or as functions that return a value. If they are used as functions, when a logical value is returned it will be .TRUE. or .FALSE., according to FORTRAN conventions. Programmers in other languages should consult Chapters 3 through 8 to see how to handle these values.

APPLIB ROUTINES

The categories of functions provided by the Applications library are:

- String Manipulation Routines
- User Query Routines
- System Information Routines
- Mathematical Routines
- Conversion Routines
- File System Routines
- Parsing Routines

The following is a detailed list of Applications subroutines by function. String manipulation routines, user query routines, and file system routines are discussed in subsequent pages of this chapter.

String Manipulation Routines

Compare two strings for equality.	CSTR$A
Compare two substrings for equality.	CSUB$A
Fill a string with a character.	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 between packed strings.	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 pathname.	TREE$A
Determine string type.	TYPE$A

User Query Routines

Prompt and read a name.	RNAM$A
Prompt and read a number (binary, decimal, octal, or hexadecimal). INTEGER*4	RNUM$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," based upon a 32-bit word size and using the Linear Congruential Method.	RAND$A
Initialize random number generator "seed."	RNDI$A

Conversion Routines

Convert a string from lowercase to uppercase or uppercase to lowercase.	CASE$A
Convert ASCII number to binary.	CNVA$A
Convert binary number to ASCII.	CNVB$A
Make a number printable if possible.	ENCD$A
Convert the DATMOD field (as returned by RDEN$$) in format DAY, MON DD YYYY	FDAT$A
Convert the DATMOD field (as returned by RDEN$$) in format DAY, DD MON YYYY.	FEDT$A
Convert the TIMMOD field (as returned by RDEN$$).	FTIM$A

File System Routines

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$A
Scan the file system structure.	TSCN$A
Check for file open.	UNIT$A

Parsing Routine

Parse PRIMOS command line.	CMDL$A

NAMING CONVENTIONS

All APPLIB and VAPPLB routines follow a consistent naming convention designed to avoid the possibility of a conflict with user-written routines and system routines. They all have a four-letter mnemonic name and the suffix $A. For example, the routine to open a temporary file is named TEMP$A.

Subroutines that are used internally by APPLIB routines have a suffix of $$A. These should not be called by programmers under ordinary circumstances.

Keys

Many routines have options which are specified by named parameter keys which all begin with the prefix A$. All parameter keys are defined in a $INSERT file named SYSCOM>A$KEYS.INS.language. The key names following the A$ prefix are three- or four-letter mnemonics specifying the allowable options for the various routines. They are INTEGER*2 data types. In addition, the FORTRAN version of this file supplies all the appropriate FUNCTION type declarations for the application routines. A complete listing of SYSCOM>A$KEYS is included at the end of this chapter. Please read the chapter on your language interface to see how to use this file.

LIBRARY IMPLEMENTATION AND POLICIES

VAPPLB and its R-mode version, APPLIB, exist as independent libraries in the UFD LIB.

The routines have been coded to make them easily callable from most other languages, including PL1G 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 application routine, but the *namlen* argument is never modified. If the *namlen* argument is not 0 or positive, 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 application routines that either accept keys as arguments, or call other routines which do, use the SYSCOM>A$KEYS file to define those keys. Also, these routines do not take advantage of any particular numerical values these keys may have, in case it should become necessary either to change these values or to add new keys with numerical values which do not fit the previous pattern. For example, there are no computed GOTOs on keys and no range checks for validity of a key. In this way, if a new SYSCOM>A$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 application 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 (LOAD). 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.

STRING MANIPULATION ROUTINES

The string manipulation routines operate on packed strings, unless stated otherwise. Most of the routines in this section require that the maximum length of a string (in characters) be passed as an argument. The maximum 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, so it may be shorter than the maximum length. (See Figure 12-1.) Since the length of a string is specified as an INTEGER*2 variable, the maximum possible length is 32767 characters.

```
 ______________________________________________
| M | Y | N | A | M | E|    |    |    |    |
 ______________________________________________
<—operational length-->

<----------maximum length------------>
```

Maximum Length and Operational Length
Figure 12-1

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:

- Maximum string-length specifications must be greater than or equal to 0. A value of 0 indicates a null or empty string.
- Substring-delimiting character positions must be greater than or equal to 0. 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 0 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..

USER QUERY ROUTINES

These routines provide a convenient means to input data from the user's terminal. Each routine can prompt the terminal user with a customized message, and then process the user's response.

FILE SYSTEM ROUTINES

The file system routines in the Applications library give the user a simple and consistent way to specify the most common file system operations. Accordingly, the Applications library 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 (Chapter 9). This library supports both Sequential Access Method (SAM) and Direct Access Method (DAM) files. There is no support for segment directory 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 filename. File units are explained in Chapter 9. Also, each routine carries the name of its function, as above, with arguments consisting of only the relevant information, usually only the file unit number. Note that all filenames, 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. Table 12-1 shows the routines available for opening.

Table 12-1
Ways to Open a File

Open name.	OPEN$A
Open funit.	OPNP$A
Open name, verify, and delay.	OPNV$A
Open funit, verify, and delay.	OPVP$A
Open scratch file.	TEMP$A

All OPEN routines can choose the file unit number upon which a file will be opened. The A$GETU key is used for this purpose and the PRIMOS file unit selected by the routine will be returned to the user (in the argument funit). If A$GETU is not used, the user must provide the routine with a usable file unit number.

Several of these subroutines have arguments called verkey, which allows verification of the validity of the file operation requested. Verification provides the following options:

1. Verify that the file is new; otherwise, verify that it is all right to modify a file which already exists.

2. Verify that the file may be modified and determine whether an existing file is to be overwritten or appended.

3. Verify that the file exists; that is, do not allow creation of a new file. Note that if the open mode is READ, this is the only possible verification option.

In case of failure of an operation, the argument wtime allows the subroutine to delay the time specified, then try again the number of times allowed by retries. Delay provides the following options:

1. If and only if the file is "IN USE", wait a supplied number of seconds (elapsed time) and try again.

2. Repeat step 1 a specified number of times.

DESCRIPTION OF SUBROUTINES

▶ CASE$A

Purpose

CASE$A is a logical function that converts a string from uppercase to lower, or from lowercase to upper. The function will be .FALSE. if length is less than 0, otherwise .TRUE..

Usage

log = CASE$A(key, string, length)

CALL CASE$A(key, string, length)

key — An INTEGER*2 option for the following conversions:

- A$FUPP — Convert all alphabetic characters in string from lowercase to uppercase.
- A$FLOW — Convert all alphabetic characters in string from uppercase to lowercase.

string — Array containing character string to be converted, packed two characters per word, any data type.

length — Length of string in characters (INTEGER*2).

APPLIB calls: GCHR$A, MCHR$A

▶ CLOS$A

Purpose

CLOS$A is a logical function that closes the file open on funit. If the operation is successful, the function is .TRUE.; otherwise, the function is .FALSE.. (This is FORTRAN logical .TRUE. and .FALSE..)

Usage

log = CLOS$A(funit)

CALL CLOS$A(funit)

funit   File unit (INTEGER*2).

APPLIB calls: None

▶ CMDL$A

Note

For Pascal and P11G programmers, CMDL$A is obsolete and has been replaced with CL$PIX. 19

Purpose

CMDL$A is a logical function for parsing a PRIMOS command line. CMDL$A 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 (words or expressions) and return information regarding each token.

Usage

```
log = CMDL$A(key,kwlist,kwindx,optbuf,buflen,option,value,kwinfo)

CALL  CMDL$A(key,kwlist,kwindx,optbuf,buflen,option,value,kwinfo)
```

key   An INTEGER*2 value specifying the following subroutine actions:

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 integer array containing control information, a table of keyword entry descriptions, and a list of default keywords. See Kwlist Format later in this chapter for a complete description.

kwindx A keyword index returned as an INTEGER*2 variable identifying the keyword in an entry. Possible values are:

< 0 Unrecognized keyword or CMDL$A was called with a key of A$RSET or A$RAWI.

0 End of line.

> 0 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. The data type does not matter.

buflen Specified length of optbuf in characters (INTEGER*2). This must be 0 or greater.

option — Returned INTEGER*2 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$NUMB — option was a number, result of numeric conversion returned in value.
- A$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 0.

kwinfo — A ten-word integer array that returns miscellaneous information and must be dimensioned in the calling program. kwinfo(1) is equal to the number of characters in optbuf and kwinfo(2) through 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.

Discussion

CMDL$A was designed to simplify the processing of a PRIMOS command line while, at the same time, providing the user with a great deal of flexibility in defining the command environment.

This routine will parse a command line, one 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:

- Integer that identifies the -keyword (kwindx)
- Text of the keyword option, if present (optbuf)
- Option type (option)

- Results of numeric conversion, if option was a number (value)
- 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 dash (-).
3. A keyword may not be a decimal number (e.g., -99).
4. Register-setting parameters (described with the R-mode EXECUTE command in the LOAD and SEG Reference Guide) are not recognized.
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 only of alphabetic characters (such as -FF) will always be interpreted as a -keyword.
10. Keyword entries in the keyword table with the same keyword index 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. (In other words, 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 uppercase 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 (n) of keyword entries in table, must be greater than 0.

Word 2 — Maximum length of keyword text in characters, must be greater than or equal to 2 and not more than 80. All keywords must have the same length and therefore it may be necessary to pad them with blanks.

Keyword Entry 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 greater than 0.

Word n+2 — Minimum number of characters in the keyword to match, including leading minus sign. The number must be no less than 2 and no greater than the maximum keyword length. A 0 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 a 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, which also constitute a valid hexadecimal number, it will be interpreted as such — for example, FACE.

A$RAWI option is the remainder of the command line after the current -keyword is read as raw text. Use of this option will turn on the end-of-line indicator in the same manner as a key of A$RAWI.

Default List

Word 1 Number (n) of default keywords, must be greater than or equal to 0.

Words 2 to n+1 List of keyword indices, previously defined in the keyword entry table, which 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. 80.
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$REQD,A$OCT,
                 * '-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
2     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
```

▶ CNVA$A

Purpose

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 is 0. Note that for decimal conversions overflow will be considered as unsuccessful, whereas for octal and hexadecimal conversions overflow is ignored.

(.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = CNVA$A(numkey, name, namlen, value)

CALL CNVA$A(numkey, name, namlen, value) numkey — An INTEGER*2 option specifying the data type of the number to be converted:

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 lengths are: binary, 31; octal, 11; decimal, 10; hexadecimal, 8. Maximum does not include leading signs or blanks.

namlen — Length of name in characters(INTEGER*2).

value — Returned converted binary value (INTEGER*4).

APPLIB calls: GCHR$A, NLEN$A

▶ CNVB$A

Purpose

CNVB$A is an INTEGER*2 or INTEGER*4 function used to convert a binary number to an ASCII digit string.

Usage

```
I*2 = CNVB$A(numkey, value, name, namlen)

CALL CNVB$A(numkey, value, name, namelen)
```

numkey — Number base to convert to (INTEGER*2); possible values are:

A$BIN	Binary number with leading blanks
A$BINZ	Binary number with leading 0s
A$DEC	Signed decimal number with leading blanks
A$DECU	Unsigned decimal number with leading blanks

A$DECZ	Signed decimal number with leading 0s
A$OCT	Octal number, leading blanks
A$OCTZ	Octal number, leading 0s
A$HEX	Hexadecimal, leading blanks
A$HEXZ	Hexadecimal, leading 0s

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 0s.

value — Binary number to be converted (INTEGER*4).

Discussion

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 0s 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 minus sign. If value is negative, binary, octal, and hexadecimal numbers will be in two's-complement form. If the number converts successfully, the function value is the number of digits and if not, it is 0.

APPLIB calls: FILL$A, MCHR$A

▶ CSTR$A

Purpose

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 (length equal to 0). Otherwise, the function will be .FALSE.. Only the operational lengths are used in the comparison. (Trailing blanks are ignored.) If the two strings are not of equal length, the result will be .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = CSTR$A(a, alen, b, blen)

a — String to be compared, packed two characters per word. Data type does not matter.

alen — Length of *a*, in characters (INTEGER*2). Length must be 0 or greater.

b — String to be compared against, packed two characters per word. Data type does not matter.

blen — Length of *b*, in characters (INTEGER*2). Length must be 0 or greater.

APPLIB calls: CSUB$A, NLEN$A

▶ CSUB$A

Purpose

CSUB$A is a logical function used to compare substrings for equality.

Usage

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 0 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 0 or greater.

bfc — First character position of substring in b (INTEGER*2).

blc — Last character position of substring in b (INTEGER*2).

Discussion

If each character in the a substring matches the corresponding character in the b substring, or both substrings are null (length equal to 0), 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.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Figure 12-2 is a representation of the arguments to CSUB$A.

```
    _____________________________________________
a   | R | O | M | A |   |   |   |   |   |   |
    _____________________________________________
      afc          alc
    <-----------------alen----------------------->


    _____________________________________________
b   | A | R | O | M | A | T | I | C |   |   |
    _____________________________________________
           bfc          blc
    <---------------blen------------------------->
```

Arguments to CSUB$A
Figure 12-2

APPLIB calls: None

▶ CTIM$A

Purpose

CTIM$A is a double-precision function that returns CPU time elapsed since login, in seconds as the function value, and as centiseconds in the cputim argument.

Usage

R*8 = CTIM$A(cputim)

CALL CTIM$A(cputim)

cputim	CPU time in centiseconds (INTEGER*4) — character string format.

Discussion

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

Purpose

DATE$A is a double-precision function that returns the date in the argument date in the form "DAY, MON DD YYYY" (for example, TUE, FEB 23 1982).

The value of the function is the date in the form "MM/DD/YY" (for example, 02/23/82). This value must be received as REAL*8.

Note that this routine is good for the period January 1, 1977 through December 31, 2076.

Usage

R*3 = DATE$A(date)

CALL DATE$A(date)

date — Date in the form DAY, MON DD YEAR. The data type does not matter as long as it is at least 16 characters long.

APPLIB CALLS: None

▶ DELE$A

Purpose

DELE$A is a logical function that deletes the file named in name. If the operation is successful, the function is .TRUE., otherwise the function is .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = DELE$A(name, namlen)

CALL DELE$A(name, namlen)

name — Filename (may be a pathname) packed two characters per word. Data type does not matter.

namlen — Length of name in characters (INTEGER*2).

APPLIB calls: TREE$A, UNIT$A, NLEN$A

▶ DOFY$A

Purpose

DOFY$A is a double-precision function that returns the day of the year in the form "DDD" in the dofy argument. 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.

Usage

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 four characters long.

APPLIB calls: None

▶ DTIM$A

Purpose

DTIM$A is a double-precision function that returns disk time since login as centiseconds is 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.

Usage

R*8 = DTIM$A(dsktim)

CALL DTIM$A(dsktim)

dsktim — Disk time in centiseconds (INTEGER*4).

APPLIB calls: None

▶ EDAT$A

Purpose

EDAT$A is a double-precision function. It returns the date in the European (military) form 'DAY, DD MON YEAR' in the argument edate (for example, TUE, 23 FEB 1982).

The value of the function is the date in the form DD.MM.YY (for example, 23.03.82). This value must be received in a REAL*8 variable. The routine is good for the period January 1, 1977 through December 31, 2076.

Usage

R*8 = EDAT$A(edate)

CALL EDAT$A(edate)

edate — Date in the form "DAY, DD MON YEAR".

Discussion

The type of the edate array does not matter as long as it is at least 16 characters long.

APPLIB calls: DATE$A

▶ ENCD$A

Purpose

ENCD$A is a logical function that converts a numeric value to a FORTRAN format.

Usage

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).

Discussion

ENCD$A 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.

Here is an explanation of the formats:

F — A number that includes a decimal fraction. The d is the number of digits after the decimal point, and w is the total number of positions (including the decimal point) in the field. The maximum is 32767.

I — An integer, with w digits. Maximum is 32767.

E — A floating point number in scientific format (xxE+yy), where xx represents the characteristic and yy is the mantissa or exponent.

Examples are:

Fw.d: 123.4

I: 12345

E: 1.23456E+99

Note that the largest value of width is 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 encoding was successful, and .FALSE. if the field was filled with asterisks. (.TRUE. and .FALSE. are the FORTRAN logical values.) Note that array is the only argument that is actually modified in the calling program.

APPLIB calls: None

▶ EXST$A

Purpose

EXST$A is a logical function that returns .TRUE. if the file exists and .FALSE. if the file does not exist or if an error was encountered. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = EXST$A(name, namlen)

name — Filename (may be a pathname) packed two characters per word. Data type does not matter.

namlen — Length of name in characters (INTEGER*2).

APPLIB calls: TREE$A, UNIT$A, NLEN$A

▶ FDAT$A

Purpose

FDAT$A is a REAL*8 function that converts the datmod field, returned as word 20 of buffer by RDEN$$, to the format DAY, MON DD YYYY (for example, TUE, FEB 23 1982).

The function value is the datmod field converted to MM/DD/YY (for example, 02/23/82). It must be received in a REAL*8 variable. The routine is good for the period January 1, 1972 to December 31, 2071.

RDEN$$ must be called before this subroutine.

Usage

CALL FDAT$A(datmod, date)

R*8 = FDAT$A(datmod, date)

datmod — Date returned by RDEN$$. This is the 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 the array is at least 16 characters long.

APPLIB calls: CNVB$A

▶ FEDT$A

Purpose

FEDT$A converts the datmod field, returned as word 20 of buffer by RDEN$$, to the format 'DAY, MON DD YEAR' in date (for example, TUE, 23 FEB 1982). The function value is datmod converted to MM.DD.YY (for example, 23.02.82). It must be received in a REAL*8 variable. The routine includes the period January 1, 1972 through December 31, 2071.

RDEN$$ must be called before this subroutine.

Usage

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 the array is at least 16 characters long.

APPLIB calls: FDAT$A

▶ FILL$A

Purpose

FILL$A is an INTEGER function that fills the name buffer with the fill character supplied. The function is INTEGER*2 or INTEGER*4, but its value is always 0.

Usage

```
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

Purpose

FSUB$A is a logical function used to fill a character substring with a specified character. The substring delimited by fchar and lchar is filled with the character specified in filchar. The string parameters are checked for validity. If an error is found, the function is .FALSE. and a message is printed. If all parameters are valid, the function will be .TRUE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

```
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

▶ FTIM$A

Purpose

FTIM$A is a REAL*4 or REAL*8 function that converts the timmod field, returned as word 21 of buffer 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.

Usage

```
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 four (INTEGER*2).

time — Array containing the 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 eight characters long.

APPLIB calls: CNVB$A

▶ GCHR$A

Purpose

GCHR$A is an INTEGER*2 or INTEGER*4 function which extracts a single character from a packed string. It is intended for use only by FORTRAN programmers. The function value will be the accessed character in FORTRAN A1 format (with blank padding on the right). The character returned will be left-justified and padded with blanks.

Usage

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).

Discussion

This routine replaces the FORTRAN statement:

```
    CHAR = FARRAY(FCHAR)
```

where FARRAY is declared LOGICAL*1 (IBM FORTRAN) or of a one-character data type.

APPLIB calls: None

▶ GEND$A

Purpose

GEND$A is a logical function that positions the file open on funit to end-of-file. If the operation is successful, the function is .TRUE., otherwise, the function is .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = GEND$A(funit)

CALL GEND$A(funit)

funit — PRIMOS file unit (INTEGER*2).

APPLIB calls: None

▶ JSTR$A

Purpose

JSTR$A is a logical function used to left-justify, right-justify, or center a string within itself.

Usage

log = JSTR$A(key, string, length)

CALL JSTR$A(key, string, length)

key — Determines direction of justification (INTEGER*2). 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). It must be greater than 0.

Discussion

The function will be .TRUE. if justification is successful, .FALSE. if the string length is less than 0 or if a bad key is used. (.TRUE. and .FALSE. are the FORTRAN logical values.)

APPLIB calls: NLEN$A, FILL$A, MSUB$A, GCHR$A

▶ LSTR$A

Purpose

LSTR$A is a logical function used to locate one string within another.

Usage

```
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	Number of characters in a (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).

Discussion

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 (length equal to 0), the function will be .FALSE. and fcp and lcp will be equal to 0. Each string is logically truncated to its operational length before the search is performed (trailing blanks are ignored).

APPLIB calls: LSUB$A, NLEN$A

▶ LSUB$A

Purpose

LSUB$A is a logical function used to locate one substring within another.

Usage

```
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).

Discussion

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 (length equal to 0), the function will be .FALSE. and fcp and lcp will be equal to 0. (.TRUE. and .FALSE. are the FORTRAN logical values.)

A representation of the arguments to LSUB$A will be found with the description of CSUB$A.

APPLIB calls: None

MCHR$A

Purpose

MCHR$A is an INTEGER function that moves a character from one packed string to another.

Usage

```
CALL MCHR$A(tarray, tchar, farray, fchar)

I*2= MCHR$A(tarray, tchar, farray, fchar)

I*4= MCHR$A(tarray, tchar, farray, fchar)
```

tarray — Returned array of characters, packed two per word, first character on the left. Data type does not matter.

tchar — Character position in tarray of received character (INTEGER*2).

farray — Source string. Data type does not matter.

fchar — Character position in farray of character to be moved (INTEGER*2).

Discussion

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 one 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

Purpose

MSTR$A is an INTEGER*2 or INTEGER*4 function used to move the source string to the destination string.

Usage

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).

Discussion

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 (length equal to 0), no characters are moved and the function value will be equal to 0.

APPLIB calls: MSUB$A

▶ MSUB$A

Purpose

MSUB$A is an integer function used to move the source substring contained in a to the destination substring contained in b.

Usage

```
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).

Discussion

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 (length equal to 0), no characters are moved and the function will be equal to 0. Otherwise it is equal to the number of characters moved (excluding blanks used for padding).

APPLIB calls: MCHR$A

▶ NLEN$A

Purpose

NLEN$A is an INTEGER*2 function that returns, as its function value, the actual length (not including trailing blanks) of the name in name.

Usage

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

▶ OPEN$A

Purpose

OPEN$A is a logical function that opens a file of the given name on funit. If the operation is successful, the function value is .TRUE., and if the operation is unsuccessful, the function value is .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = OPEN$A(opnkey+typkey+untkey, name, namlen, unit)

CALL OPEN$A(opnkey+typkey+untkey, name, namlen, unit)

opnkey — INTEGER*2:

A$READ — Open for reading.

A$WRIT — Open for writing.

A$RDWR — Open for reading and writing.

typkey INTEGER*2:

A$SAMF SAM file

A$DAMF DAM file

untkey INTEGER*2:

A$GETU Choose a PRIMOS file unit number to be returned in funit. Omission of this key requires that the routine be provided with a unit number (INTEGER*2).

name File name (may be a pathname) packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2).

funit PRIMOS file unit. This value is 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

Purpose

OPNP$A is a logical function that gets a name from the user and opens it on funit. 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.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

```
log = OPNP$A(msg, msglen, opnkey+typkey+untkey, name, namlen, funit)

CALL  OPNP$A(msg, msglen, opnkey+typkey+untkey, name, namlen, funit)
```

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 INTEGER*2:

- A$READ Open for reading.
- A$WRIT Open for writing.
- A$RDWR Open for reading and writing.

typkey INTEGER*2:

- A$SAMF SAM file
- A$DAMF DAM file

untkey INTEGER*2:

- A$GETU Choose a PRIMOS file unit number to be returned in funit. Omission of this key requires that the routine be provided with a unit number.

name Filename (may be a pathname) packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2).

funit PRIMOS file unit returned if untkey is 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

Purpose

OPNV$A is a logical function that opens a file of the given name on funit. Note that the functions of verification and delay as described here and in the section FILE SYSTEM ROUTINES above are independent of each other.

Usage

```
log = OPNV$A(opnkey+typkey+untkey, name, namlen, funit, verkey, wtime,
          retries)

CALL  OPNV$A(opnkey+typkey+untkey, name, namlen, funit, verkey,
            wtime, retries)
```

opnkey INTEGER*2:

A$READ Open for reading.

A$WRIT Open for writing.

A$RDWR Open for reading and writing.

typkey INTEGER*2;

A$SAMF SAM file

A$DAMF DAM file

untkey INTEGER*2:

A$GETU Choose a PRIMOS file unit number to be returned in funit. Omission of this key requires that the routine be provided with a unit number.

name Filename (may be a pathname) packed two characters per word. Data type does not matter.

namlen Length of name in characters (INTEGER*2). If namlen is 0 or less, the function value is .FALSE..

funit PRIMOS file unit returned if untkey =A$GETU. If not, user must provide a legal file number in this argument (INTEGER*2).

verkey INTEGER*2:

A$NVER No verification.

A$VNEW Verify new or ask if OK to modify 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).

retries Number of times to retry if FILE IN USE (INTEGER*2).

Discussion

If wtime and retries are specified as nonzero 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 retries is initially 0 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 not requested (verkey = A$NVER), OPNV$A is identical in function to OPEN$A. If verification is requested (verkey other than A$NVER), the following actions will be taken according to the value of verkey:

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.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

▶ OPVP$A

Purpose

OPVP$A is a logical function that gets a filename from the user and opens it on funit. Note that the functions of verification and delay as described below, and in the section FILE SYSTEM ROUTINES above, are independent of each other.

Usage

```
log = OPVP$A(msg, msglen, opnkey+typkey+untkey, name, namlen, funit,
             verkey, wtime, retries)

CALL  OPVP$A(msg, msglen, opnkey+typkey+untkey, name, namlen, funit,
             verkey, wtime, retries)
```

msg — Array containing prompt message, packed two characters per word. Data type does not matter.

msglen — Length of msg in characters (INTEGER*2).

opnkey — INTEGER*2:

A$READ	Open for reading.
A$WRIT	Open for writing.
A$RDWR	Open for reading and writing.

typkey — INTEGER*2:

A$SAMF	SAM file
A$DAMF	DAM file

untkey — INTEGER*2:

A$GETU	Choose a file unit number to be returned in funit. Omission of this key requires that the routine be provided with a unit number.

name — Array containing filename (may be pathname), packed two characters per word. Data type does not matter.

namlen — Length of name in characters (INTEGER*2). If namlen is 0 or less, the function value is .FALSE..

funit — File unit returned if untkey = A$GETU. If not, user must provide a legal file unit in this argument (INTEGER*2).

verkey — INTEGER*2:

A$NVER — No verification.

A$VNEW — Verify new file or ask if OK to modify 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).

retries — Number of times to retry if FILE IN USE (INTEGER*2).

Discussion

If wtime and retries are specified as nonzero 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 attempts. If the number or retries expires, or if either wtime or retries is initially 0 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, according to the value of verkey:

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.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

▶ POSN$A

Purpose

POSN$A is a logical function that 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.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = POSN$A(poskey, funit, pos)

CALL POSN$A(poskey, funit, pos)

poskey — INTEGER*2:

A$ABS	Absolute position
A$REL	Relative position

funit — PRIMOS file unit (INTEGER*2)

pos — Postion (relative or absolute) (INTEGER*4)

APPLIB calls: None

▶ RAND$A

Purpose

RAND$A is a random-number generator.

Usage

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).

Discussion

RAND$A is a double-precision function that updates a seed to a new seed based upon the following 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.

For examples, see Chapters 4 through 8.

APPLIB calls: None

▶ RNAM$A

Purpose

RNAM$A is a logical function that prints the supplied message prompt and appends a colon (:) 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.. (.TRUE. and .FALSE. are the FORTRAN logical values.) The caller should be aware that COMANL and RDTK$$ (Chapter 9) are called to read the user response, and therefore the previous command line entered is unavailable.

Usage

```
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 — An INTEGER*2 option key. Keys cannot be combined.

- A$FUPP — Force uppercase.
- A$UPLW — Do not force uppercase.
- A$RAWI — Read line as raw uninterpreted text.

name — Returned name, packed two characters per word. Data type does not matter.

namlen — Length of name buffer in characters (maximum 80) (INTEGER*2).

APPLIB calls: None

▶ RNDI$A

Purpose

RNDI$A is a double-precision function that 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.

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.

Usage

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

▶ RNUM$A

Purpose

RNUM$A is a logical function used to accept numeric data from the user terminal.

Usage

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 — An INTEGER*2 key specifying the data type to be verified:

A$DEC	Decimal
A$BIN	Binary
A$OCT	Octal
A$HEX	Hexadecimal

value — Returned value (INTEGER*4).

Discussion

The routine prints the user-supplied message and appends the colon (:) 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 will be 0. If a legal number is provided, the function will be .TRUE. and the value will be returned in value. (.TRUE. and .FALSE. are the FORTRAN logical values.)

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 two's complement, but the same as a negative decimal number.

The caller should be aware that COMANL and RDTK$$ (Chapter 10) are called to read the user response, and therefore the previous command line is unavailable.

Examples of calls to RNUM$A are given in Chapters 3 through 8. The operation of this subroutine is shown in Figure 12-3.

APPLIB calls: None

▶ RPOS$A

Purpose

RPOS$A is a logical function that returns the current absolute position of the file open on unit. If the operation is successful, the function is .TRUE.; otherwise the function is .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

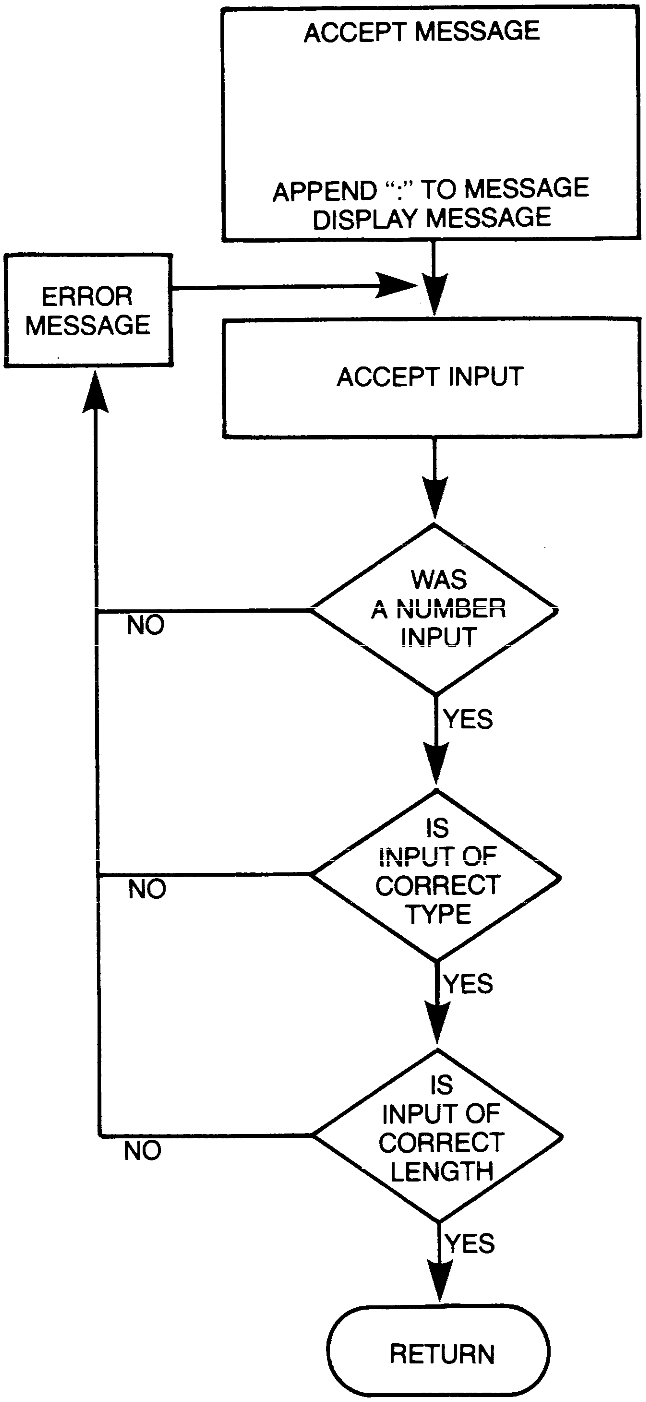
Usage

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



How RNUM$A Works
Figure 12-3

▶ RSTR$A

Purpose

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 0, the function will be .FALSE., otherwise the function will be .TRUE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

```
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. Figure 12-4 shows the results of two calls to RSTR$A.

```
 ___________________________
| 1 | 2 |   | 4 | 5 | 6 |
 ___________________________
<------string--------->

 ___________________________
| 4 | 5 | 6 | 1 | 2 |   |
 ___________________________
after RSTR$A(string, 6, -3)

 ___________________________
| 1 | 2 | 4 | 5 | 6 |   |
 ___________________________
after RSTR$A(string, 6, 2)
```

Use of RSTR$A
Figure 12-4

Example

The following example performs the operations diagrammed above.

```
OK, SLIST ROTATE.COBOL

        IDENTIFICATION DIVISION.
        PROGRAM-ID. ROTATE.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 STRING1  PIC X(32) VALUE '12 456
        01 LENGTH   COMP.
        01 CNT      COMP.
        PROCEDURE DIVISION.
        001-BEGIN.
            MOVE 6 TO LENGTH.
            MOVE -3 TO CNT.
            CALL 'RSTR$A' USING STRING1, LENGTH, CNT.
            EXHIBIT STRING1.
            MOVE 2 TO CNT.
            CALL 'RSTR$A' USING STRING1, LENGTH, CNT.
            EXHIBIT STRING1.
            STOP RUN.

OK, COBOL ROTATE
 Phase I
 Phase II
 Phase III
 Phase IV
 Phase V
 Phase VI
```

```
No Errors, No Warnings, Prlme V-Mode COBOL, Rev 18.4 <ROTATE>

OK, SEG -LOAD
[SEG rev 18.4]
$ LO ROTATE
$ LI VCOBLB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC
STRING1 = 45612
STRING1 = 12456
OK,
```

▶ RSUB$A

Purpose

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. If there is an error, a message is printed and the function will be .FALSE.. If no error occurs, the function will be .TRUE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

```
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).

Discussion

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

▶ RWND$A

Purpose

RWND$A is a logical function that rewinds the file open on unit. If the operation is successful, the function is .TRUE.; otherwise the function is .FALSE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = RWND$A(unit)

CALL RWND$A(unit)

unit PRIMOS file unit (INTEGER*2)

APPLIB calls: None

▶ SSTR$A

Purpose

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 0, an error message is printed, the function is .FALSE., and no characters are shifted. If no error occurs, the function is .TRUE.. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

```
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 0 (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 FORTRAN A1 format (two characters per word and blank-padded on the right). Data type does not matter.

APPLIB calls: FSUB$A, MCHR$A, NLEN$A

► SSUB$A

Purpose

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. An error will cause a message to be printed and the function will be .FALSE.. If no error occurs, the function will be .TRUE.. (.TRUE. and .FALSE. are the FORTRAN logical values.) If the substring is null, or length is equal to 0, there will be no shift.

Usage

```
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 (two characters per word and right-padded with blanks). Data type does not matter.

APPLIB calls: FSUB$A, MCHR$A

▶ TEMP$A

Purpose

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 four-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. (These are the FORTRAN logical values.)

Usage

```
log = TEMP$A(typkey+untkey, name, namlen, funit)

CALL  TEMP$A(typkey+untkey, name, namelen, funit)
```

typkey — INTEGER*2:

- A$SAMF — SAM file
- A$DAMF — DAM file

untkey — INTEGER*2

- A$GETU — Choose a file unit number to be returned in funit. Omission of this key requires that the routine be provided with a unit number (INTEGER*2).

name — Returned name (six characters, packed two characters per word). Data type does not matter.

namlen — Length of name buffer in characters (must be at least six) (INTEGER*2).

funit — File unit (INTEGER*2).

APPLIB calls: FILL$A

▶ TIME$A

Purpose

TIME$A is a double-precision function that 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.

Usage

```
R*8 = TIME$A(time)

CALL  TIME$A(time)
```

time — Time of day in the form HH:MM:SS, packed two characters per word. Data type does not matter as long as it is at least eight characters long.

APPLIB calls: None

▶ TREE$A

Purpose

TREE$A is a logical function that scans a file name and determines if it is a pathname. If it is a pathname, the function is .TRUE. and if not, it is .FALSE.. In addition, the location of the final name (or entire name if not part of a pathname) may be determined from the values returned in fst and flen. Note that if the name is empty, fst and flen are both 0.

Usage

log = TREE$A(name, namlen, fst, flen)

name — Array containing filename, packed two characters per word (input). Data type does not matter.

namlen — Length of name in characters (INTEGER*2 — input).

fst — Character position in name of first character in final name (INTEGER*2 — returned).

flen — Length of final file name in characters (INTEGER*2 — returned).

APPLIB calls: GCHR$A, NLEN$A

Figure 12-5 is a representation of the arguments to TREE$A.

Example

```
OK, SLIST TREE.COBOL

        IDENTIFICATION DIVISION.
        PROGRAM-ID. TREE.
        ENVIRONMENT DIVISION.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01 NAME PIC X(32) VALUE SPACES.
        01 NAMLEN COMP.
        01 FSTART COMP.
        01 FLEN COMP.
        01 ASCIILEN PIC S99.
        PROCEDURE DIVISION.
        001-BEGIN.
            DISPLAY 'ENTER FILENAME'.
            ACCEPT NAME.
            DISPLAY 'ENTER LENGTH OF NAME'.
            ACCEPT ASCIILEN.
            MOVE ASCIILEN TO NAMLEN.
            CALL 'TREE$A' USING NAME, NAMLEN, FSTART, FLEN.
            EXHIBIT NAME.
            EXHIBIT NAMLEN.
            EXHIBIT FSTART.
            EXHIBIT FLEN.
            STOP RUN.

OK, SEG TREE
ENTER FILENAME
ANNE>SUBS>TREE
```

```
ENTER LENGTH OF NAME
14
NAME= ANNE>SUBS>TREE
NAMLEN= 00014+
FSTART= 00011+
FLEN= 00004+
OK,
```

```
 ________________________________________________________________
 |A|N|N|E|>|D|A|T|A|>|S|A|M|D|A|T|A|>|H|O|U|R|S|W|O|R|K|E|D|
 ________________________________________________________________
 <--------------------------nameln---------------------------->
                                      <----------flen--------->
                                      fst
```

Arguments to TREE$A
Figure 12-5

▶ TRNC$A

Purpose

TRNC$A is a logical function that truncates the file open on funit. If the operation is successful, the function is .TRUE.; otherwise the function is .FALSE. (These are the FORTRAN logical values.)

Usage

```
log =   TRNC$A(funit)

CALL  TRNC$A(funit)
```

funit   PRIMOS file unit (INTEGER*2)

APPLIB calls: None

▶ TSCN$A

Purpose

TSCN$A is a logical function that scans the file system tree structure (starting with the home UFD). It uses the file subroutines RDEN$$ and SGDR$$ to read UFD and segment directory entries into the entry array.

Usage

```
log= TSCN$A(key, funits, entry, maxsiz, entsiz, maxlev, lev, code)

CALL TSCN$A(key, funits, entry, maxsiz, entsiz, maxlev, lev, code)
```

key INTEGER*2:

A$TREE	Scan full tree.
A$NUFD	Do not scan sub-UFDs.
A$NSEG	Do not scan segment directories.
A$CUFD	Scan current UFD only.
A$DLAY	Pause when popping up to directory.

funits Array of unit numbers maxlev long (INTEGER*2).

entry Array maxsiz * maxlev long (INTEGER*2).

> **Caution**
>
> This two-dimensional array may be passed from a FORTRAN program only.

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 Return code (INTEGER*2).

APPLIB calls: None

Discussion

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=0), 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 sub-UFD, following the next call to TSCN$A lev will be 2, and entry(1,2) will contain the entry for the first file in the sub-UFD. Thus, for the UFD represented in Figure 12-6, TSCN$A in a loop would return the names in the order shown in Figure 12-7.

The values of key (INTEGER*2) have the following meanings:

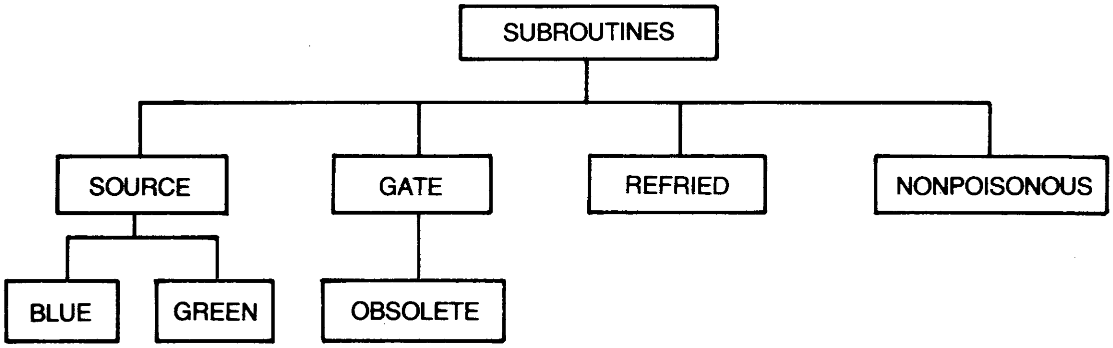
A$TREE — All entries in the directory structure are returned up to maxlev levels deep. (Levels below level maxlev are ignored.)

A$NUFD — When a sub-UFD is encountered (in the home UFD), its entry is returned, but no files under that sub-UFD are returned. In the absence of segment directories, this effectively limits the 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 a tree-delete function since a directory cannot be deleted until it has been emptied.)



A UFD to be Searched by TSCN$A
Figure 12-6

```
SOURCE
SOURCE > BLUE
SOURCE > GREEN
GATE
GATE > OBSOLETE
NONPOISONOUS
REFRIED
OK,
```

Result of TSCN$A Sample Program on Figure 12-6
Figure 12-7

The following items should be considered when using TSCN$A:

1. For the first call of TSCN$A, lev should be equal to 0. Thereafter it should not be modified until EOF is reached on the top level UFD at which point lev will be reset to 0.

2. The entries in the entry array are in RDEN$$ format. For entries inside a segment directory, all information from the directory entry is first copied down a level. Entry(2,lev) is set to 0 and entry(3,lev) is then set to a 16-bit entry number. For nested segment directories, the type field of the entry is set appropriately by opening the file with SRCH$$. (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(20,1) — 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 the RDEN$$ and SGDR$$ read-position and set-position functions carefully, it is possible to reuse file units dynamically.

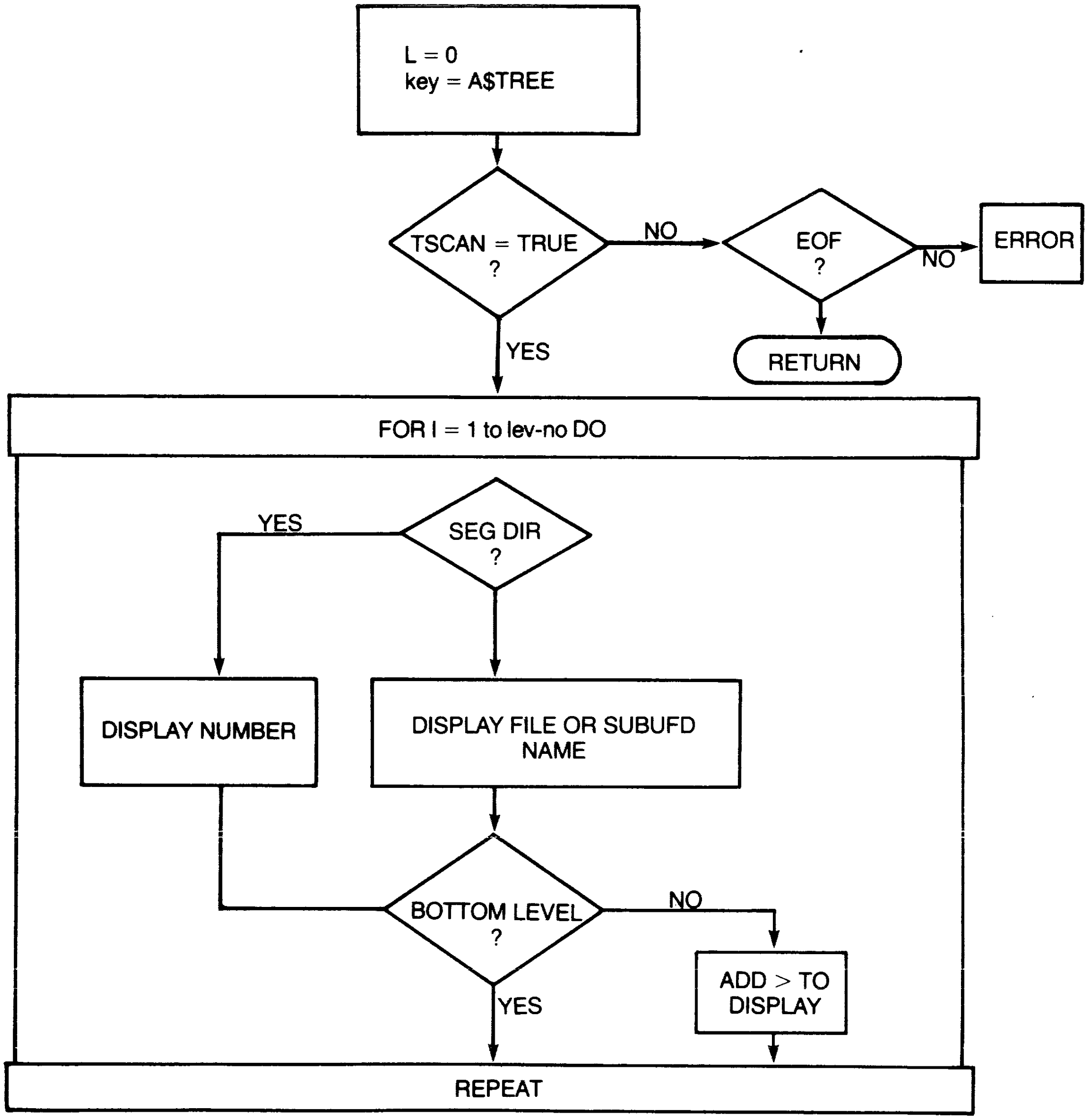
6. TSCN$A returns .TRUE. until a nonzero file system code is returned or until E$EOF is returned with lev=0 (top level). E$EOF on lower levels of the structure is suppressed, and code is returned as 0.

7. TSCN$A requires owner rights in the home UFD.

Example

The following FORTRAN program illustrates how TSCN$A can be used to perform a directory LISTF. Figures 12-6 and 12-7 show the results of the program run in a sample directory. Figure 12-8 diagrams how the program works.

```
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>A$KEYS.INS.FTN
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 PATHNAME
        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)/* PATHNAME SEPARATOR
200     CONTINUE
      CALL TONL
      GOTO 100
      END
```



Using TSCN$A to List Files on Directories
(See sample program.)
Figure 12-8

▶ TYPE$A

Purpose

TYPE$A is a logical function that tests a character string to determine if it can be interpreted as the type specified by key.

Usage

log = TYPE$A(key, string, length)

key — String type to be tested for (INTEGER*2). 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).

Discussion

A string is interpreted as a name if it contains at least one alphabetic or special character other than a leading plus or minus; a binary number if it contains only the digits 0 through 1; a decimal number if it contains only the digits 0 through 9. It is an octal number if it contains only the digits 0 through 7, and is hexadecimal if it contains only the digits 0 through 9 and the characters A through F (uppercase only). A number may have a leading sign and any number of blanks between the sign and the first digit. However, embedded 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 (length equal to 0) will return a function value of .TRUE. only if key is A$NAME.

APPLIB calls: GCHR$A, NLEN$A

▶ UNIT$A

Purpose

UNIT$A is a logical function that returns .TRUE. if a file unit is open and .FALSE. if it is not open. (.TRUE. and .FALSE. are the FORTRAN logical values.)

Usage

log = UNIT$A(funit)

funit PRIMOS file unit (INTEGER*2)

APPLIB calls: None

▶ YSNO$A

Purpose

YSNO$A is a logical function that 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 one or two characters.

Usage

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).

msglen   Message length in characters (INTEGER*2).

defkey   An INTEGER*2 key specifying the default:

A$NDEF   No default accepted.

A$DNO   Default is "NO".

A$DYES   Default is "YES".

APPLIB calls: None

Example

```
OK, SLIST YESNO1.PASCAL

program main;
{                                                              }
{ FORTRAN logicals are incompatible with Pascal boolean data types.}
{ Therefore, interfacing to the applications library from Pascal  }
{ can be a problem. The following program shows the easiest way to }
{ determine True and False when calling FORTRAN subroutines with   }
{ logicals.                                                        }
{                                                                  }
{ Note: This program assumes that the type of logical returned is  }
{       a LOGICAL*2, and only occupies two bytes of memory.        }
{                                                                  }
const
%INCLUDE 'SYSCOM>A$KEYS.INS.PASCAL';

type
     string8 = array [1 .. 8] of char;

     string16 = array[1 ..16] of char;

var
     msg : string16;
     date: string16;
     time: string8;

function ysno$a(var s : char;     {Pass by ref, first loc of the msg}
                    l : integer; {Pass by value, length of msg      }
                    k : integer) {Pass by value, default keys       }
               :integer; extern; {Returns FORTRAN logical as integer}

begin
     writeln;

     msg := 'Yes | No        ';
     if ord( True ) = ysno$a(msg[1],8, a$ndef)
       then
```

```
             writeln('Ok!')
         else
             writeln('Absolutely NO!');
   end.
```

This program, stored as YESNO1.PASCAL, may be compiled, loaded, and executed with the following dialogue.

```
    OK, PASCAL YESNO1
    0000 ERRORS (PASCAL-REV 19.0)
    OK, SEG -LOAD
    [SEG rev 19.0]
    $ LO YESNO1
    $ LI PASLIB
    $ LI VAPPLB
    $ LI
    LOAD COMPLETE
    $ EX

    Yes | No? YES
    Ok!
    OK, SEG YESNO1

    Yes | No? NO
    Absolutely NO!
    OK,
```

FORMAT SUMMARY

Below is a brief summary of the calling sequences for all the VAPPLB and APPLIB routines. The type codes are defined as:

Type Code	Description
L	LOGICAL
I	INTEGER*2 or INTEGER*4
I*2	INTEGER*2
R	REAL
DP	DOUBLE PRECISION

Group	Name	Type	Arguments
File System	TEMP$A	L	(TYPKEY,NAME,NAMLEN,FUNIT)
	OPEN$A	L	(OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN,FUNIT)
	OPNP$A	L	(MSG,MSGLEN,OPNKEY+TYPKEY+UNTKEY,NAME, NAMLEN,FUNIT)
	OPNV$A	L	(OPNKEY+TYPKEY+UNTKEY,NAME,NAMLEN,FUNIT, VERKEY,WTIME,RETRYS)
	OPVP$A	L	(MSG,MSGLEN,OPNKEY+TYPKEY+UNTKEY,NAME, NAMLEN,FUNIT,VERKEY,WTIME,RETRYS)
	CLOS$A	L	(FUNIT)
	RWND$A	L	(FUNIT)
	GEND$A	L	(FUNIT)
	TRNC$A	L	(FUNIT)
	DELE$A	L	(NAME,NAMLEN)
	EXST$A	L	(NAME,NAMLEN)
	FUNIT$A	L	(FUNIT)
	RPOS$A	L	(FUNIT,POS)
	POSN$A	L	(POSKEY,FUNIT,POS)
	TSCN$A	L	(KEY,FUNITS,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

This is a listing of the file SYSCOM>A$KEYS, as needed for FORTRAN programs. Pascal and PL1G programmers should use the A$KEYS.INS.language file that is applicable.

This listing uses decimal values for keys. The listings from the SYSCOM UFD use octal values.

```
C A$KEYS.INS.FTN, APPLIB>SOURCE, TRANSLATOR DEPT, 05/29/81
      /************************************************************** */
      /*                                                              */
      /*            KEY DEFINITIONS (TABSET 6 11 28 69)               */
      /*                                                              */
      /*********** OPEN$A, OPNP$A, OPNV$A, TEMP$A  ************        */
      /********************* OPNKEY ************************          */
         A$READ = 1,      /* READ                                     */
         A$WRIT = 2,      /* WRITE                                    */
         A$RDWR = 3,      /* READ/WRITE                               */
      /*               ****** TYPKEY ******                           */
         A$SAMF = 0,      /* OPEN NEW SAM FILE                        */
         A$DAMF = 1024 ,  /* OPEN NEW DAM FILE                        */
      /*               ****** UNTKEY ******                           */
         A$GETU = 16348, /* OPEN AND RETURN UNIT                      */
      /*               ****** VERKEY ******                           */
         A$NVER = 1,      /* NO VERIFICATION                          */
         A$VNEW = 2,      /* VERIFY NEW FILE (OK TO MODIFY)           */
         A$OVAP = 3,      /* A$VNEW + OVERWRITE/APPEND OPTION         */
         A$VOLD = 4,      /* VERIFY OLD FILE (DO NOT CREATE NEW)      */
      /*                                                              */
      /********************* RPOS$A ******************************** */
      /*               ****** POSKEY ******                           */
         A$ABS  = 1,      /* ABSOLUTE POSITION                        */
         A$REL  = 2,      /* RELATIVE POSITION                        */
      /*                                                              */
      /********************* YSNO$A ******************************** */
      /*               ****** DEFKEY ******                           */
         A$NDEF = -1,     /* NO DEFAULT                               */
         A$DNO  = 0,      /* DEFAULT = 'NO'                           */
         A$DYES = 1,      /* DEFAULT = 'YES'                          */
      /*                                                              */
      /********************* RNUM$A ******************************** */
      /********************* CNVA$A ******************************** */
      /*               ****** NUMKEY ******                           */
         A$DEC  = 1,      /* DECIMAL                                  */
         A$OCT  = 2,      /* OCTAL                                    */
         A$HEX  = 3,      /* HEXADECIMAL                              */
         A$BIN  = 9,      /* BINARY                                   */
      /*                                                              */
      /*                                                              */
      /********************* CNVB$A ********************              */
      /*               ****** NUMKEY ******                           */
      /* A$DEC  = 1,      /* DECIMAL,LEFT PADDED WITH BLANKS          */
/* A$OCT  = 2,      /* OCTAL, LEFT PADDED WITH BLANKS            */
/* A$HEX  = 3,      /* HEXADECIMAL, LEFT PADDED WITH BLANKS      */
/* A$BIN  = 9,      /* BINARY, LEFT PADDED WITH BLANKS           */
   A$DECZ = 4,      /* DECIMAL, LEFT PADDED WITH ZEROS           */
   A$OCTZ = 5,      /* OCTAL, LEFT PADDED WITH ZEROS             */
   A$HEXZ = 6,      /* HEXADECIMAL, LEFT PADDED WITH ZEROS       */
   A$DECU = 7,      /* UNSIGNED DECIMAL, LEFT PADDED WITH        */
/*                     BLANKS                                    */
   A$BINZ = 8,      /* BINARY, LEFT PADDED WITH ZEROS            */
/*                                                               */
/*                                                               */
/******************** CMDL$A ********************************** */
/*            ****** KEY    ******                               */
/* A$READ = 1,      /* READ NEXT ENTRY IN COMMAND LINE           */
   A$NEXT = 2,      /* READ FIRST ENTRY IN NEXT LINE             */
   A$RSET = 3,      /* RESET TO BEGINNING OF COMMAND LINE        */
/* A$RAWI = 4,      /* READ REMAINDER OF LINE AS RAW TEXT        */
   A$NKWL = 5,      /* ACCEPT NEW KEYWORD LIST                   */
   A$RCMD = 6,      /* FIRST TOKEN IS COMMAND (NO '-')           */
/*            ****** OPTYPE ******                               */
/* A$DEC  = 1,      /* DECIMAL OPTION                            */
/* A$OCT  = 2,      /* OCTAL OPTION                              */
/* A$HEX  = 3,      /* HEXADECIMAL OPTION                        */
/* A$RAWI = 4,      /* OPTION IS RAW TEXT                        */
   A$NDEC = 5,      /* NAME OR DECIMAL OPTION                    */
   A$NOCT = 6,      /* NAME OR OCTAL OPTION                      */
   A$NHEX = 7,      /* NAME OR HEXADECIMAL                       */
   A$NAME = 8,      /* NAME                                      */
/* A$BIN  = 9,      /* BINARY OPTION                             */
   A$NBIN =10,      /* NAME OR BINARY OPTION                     */
/*            ****** OPTION ******                               */
   A$NONE = 0,      /* NO OPTION PRESENT OR NULL OPTION          */
/* A$NAME = 8,      /* OPTION IS A NAME                          */
   A$NUMB = 9,      /* OPTION IS A NUMBER (DIGIT STRING)         */
   A$NOVF = 10,     /* NUMERIC OVERFLOW                          */
/*            ****** STATUS ******                               */
/* A$NONE = 0,      /* NO OPTION TO FOLLOW KEYWORD               */
   A$OPTL = 1,      /* OPTION MAY OR MAY NOT FOLLOW KEYWORD      */
   A$REQD = 2,      /* OPTION MUST FOLLOW KEYWORD                */
/*                                                               */
/******************** RNAM$A ********************************** */
/*            ****** NAMKEY ******                               */
   A$FUPP = 1,      /* FORCE UPPER CASE                          */
   A$UPLW = 2,      /* READ UPPER AND LOWER CASE                 */
   A$RAWI = 4,      /* READ REST OF LINE                         */
/*                                                               */
/*                                                               */
/******************** TSCN$A ********************************** */
/*            ****** KEY    ******                               */
   A$TREE = 1,      /* ALL ENTRIES IN A TREE                     */
   A$NUFD = 2,      /* DO NOT SCAN SUBUFDS                       */
   A$NSEG = 3,      /* DO NOT SCAN SEGDIRS                       */
   A$CUFD = 4,      /* DO NOT SCAN SUBUFDS OR SEGDIRS            */
   A$DLAY = 5,      /* STAY AT DIRECTORY WHEN GOING UP TREE      */
   A$BACK = 6,       /* BACK UP ONE LEVEL (FOR ERROR HANDLING)     */
/*                                                                 */
/********************** JSTR$A *********************************** */
/*               ****** KEY    ******                              */
   A$RGHT = 1,       /* RIGHT JUSTIFY                              */
   A$LEFT = 2,       /* LEFT JUSTIFY                               */
   A$CNTR = 3,       /* CENTER                                     */
/*                                                                 */
/********************** CASE$A *********************************** */
/*               ****** KEY    ******                              */
/* A$FUPP = 1,       /* FORCE UPPER CASE                           */
   A$FLOW = 5        /* FORCE LOWER CASE                           */
/*                                                                 */
/********************** TYPE$A *********************************** */
/*               ****** KEY    ******                              */
/* A$BIN  = 9,       /* BINARY NUMBER                              */
/* A$DEC  = 1,       /* DECIMAL NUMBER                             */
/* A$OCT  = 2,       /* OCTAL NUMBER                               */
/* A$HEX  = 3,       /* HEXADECIMAL NUMBER                         */
/* A$NAME = 8        /* NAME                                       */
/*                                                                 */
/***************************************************************** */
LIST
```

13
Sort Libraries

SORT SUBROUTINES OVERVIEW

PRIMOS contains many routines for performing disk or internal sorts. The subroutines are contained in the four libraries described below. A detailed description of each subroutine follows later in this chapter.

VSRTLI is the V-mode sort library. It contains the disk sort routines ASCSRT (also called ASCS$$), which can sort on five key types and can merge sorted files, and SUBSRT, which will sort one file on an ASCII key. These routines handle larger records and more key types and record types than the R-mode version. VSRTLI also has a set of cooperating subroutines which provide for the user's own input and output procedures. Their strategy is described in the sections on COOPERATING MERGE SUBROUTINES and COOPERATING SORT SUBROUTINES below.

SRTLIB is the R-mode sort library. It contains two subroutines that perform a disk sort operation. SUBSRT will sort one file on multiple ASCII keys; ASCS$$ can sort on five key types and can also merge sorted files.

The VMSORT library contains several in-memory sort subroutines (heapsort, bubble, partition exchange, radix exchange, straight insertion, binary search, and diminishing increment). It also has a binary-search and table-building subroutine.

MSORTS is the R-mode version of VMSORT.

Table 13-1 shows the subroutines by function. Table 13-2 shows which subroutines are in each sort library.

Table 13-1
Sort Routines by Function

Sort one file on ASCII key(s).	SUBSRT
Sort or merge sorted files (multiple key types).	ASCSRT, ASCS$$
Merge sorted files.	MRG1$S
Return next merged record to sort.	MRG2$S
Close merged input files.	MRG3$S
Sort one or several input files.	SRTF$S
Prepare sort table and buffers.	SETU$S
Get input records.	RLSE$S
18.1 Sort tables prepared by SETU$S.	CMBN$S
Get sorted records.	RTRN$S
Close all sort units.	CLNU$S
Heap sort.	HEAP
Partition exchange.	QUICK
Dimishing increment.	SHELL
Radix exchange.	RADXEX
Insertion sort.	INSERT
Bubble sort.	BUBBLE
Binary search or build binary table.	BNSRCH

Table 13-2
Sort Subroutines by Library

SRTLIB	VSRTLI	MSORTS	VMSORT
SUBSRT	SUBSRT	HEAP	HEAP
ASCS$$	ASCS$$	QUICK	QUICK
	ASCSRT	SHELL	SHELL
	SRTF$S	RADXEX	RADXEX
	SETU$S	INSERT	INSERT
	RLSE$S	BUBBLE	BUBBLE
	CMBN$S	BNSRCH	BNSRCH
	RTRN$S		
	CLNU$S		
	MRG1$S		
	MRG2$S		
	MRG3$S		

18.1

Record Types

The following record types are handled by the VSRTLI 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. Files containing records of this type are also called *binary files* (not the same as object files produced by a compiler).

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 (ED), the character must be included in the length.)

Default Record Type: The default depends upon the key types specified. (See Key Definitions, below.) The input type defaults to variable length if the key specifies a single-precision (16-bit) integer, double-precision (32-bit) integer, or single- or double-precision real number. Otherwise, the default is compressed source. If the output type is not specified, it is assumed to be the same as the input type.

SRTLIB routines use only compressed-source and variable records.

Note

> 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 32760 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.

Collating Sequence

18.1 ASCII keys may be sorted using the EBCDIC rather than the ASCII collating sequence. This option is specified in the spcls(2) parameter of SRTF$S and SETU$S.

Key Definitions

A sort key is a portion of the record, called the record field, on which the records are to be sorted. Each key must start and end on a byte boundary. An improperly defined key (e.g., with record length 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 64 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: 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 as long as 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
0	0,-,+,{	};-
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: 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	±(10**-38 to 10**38)
Double-precision real	8	±(10**-9902 to 10**9825)
18.1 Unsigned integer	2	0 to 65535

Packed Decimal Keys: Stored two digits per byte. The last byte contains the final digit plus sign. A negative field has a hex D in the sign nibble. All other four-bit combinations in the sign nibble represent a positive sign. A packed field must have an odd number of digits and may have up to 63 digits plus sign.

Arguments

Numeric parameters are INTEGER*2 unless otherwise noted. Names are received as integer arrays, so the data type does not matter in the calling program.

Tag Sort

When a sort cannot be done completely in the memory allocated, it creates temporary work files in which it stores sorted pieces of the
18.1 data. These sorted pieces are then merged to create the output file.

A tag sort will store the input records separate from the key data. After all the keys have been sorted and merged, the corresponding records are then located and output. The more records there are, the longer this may take, and so this last phase may be time-consuming for a very large file.

A nontag sort will store each input record with its sort key. This eliminates the search for each record after merging, but requires more disk space. However, a nontag sort will not always be faster, since more I/O must be done to merge records and keys than to merge keys only.

Some selection criteria, in probable order of importance, are:

- If disk space is a problem, use a tag sort.
- If the input file is small, it doesn't matter. 18.1
- If the input file is big, use a nontag sort.
- If the input file is partially ordered, use a nontag sort.
- If the input file is not ordered, use a tag sort.

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. 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. Up to 64 keys may be specified. The maximum record length is 32760 bytes. 18.1

▶ SUBSRT

Purpose

SUBSRT is used to sort a single input file containing compressed source records on ASCII keys in ascending order. Maximum record length is 32760 bytes (characters); maximum key length is 312 characters. 18.1

Usage

CALL SUBSRT(path-1,len-1,path-2,len-2,numkey,nstart,nend,npass,nitem)

path-1 Input pathname.

len-1 Length of input pathname in characters, up to 80.

path-2 Output pathname.

len-2 Length of output pathname in characters, up to 80.

numkey Number of keys (pairs of starting and ending columns) — starting and ending bytes if binary. Maximum is 64, default is 1. 18.1

nstart	Vector containing starting columns/bytes of keys (must be greater than or equal to 1).
nend	Vector containing ending columns/bytes of keys. Each ending column must be no greater than linsiz.
npass	Number of passes (returned).
nitem	Number of items returned in output file (INTEGER*4).

▶ ASCSRT (ASCS$$)

Purpose

ASCSRT (which can also be called as ASCS$$) is the V-mode subroutine that handles larger records and more types of sort key fields than the R-mode version. Maximum record length is 32760 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.

Usage

```
CALL {ASCS$$} (path-1,len-1,path-2,len-2,numkey,nstart,nend,npass,
     {ASCSRT}  nitem,nrev,ispce,mgcnt,mgbuff,len,LOC(buffer),msize,
               ntype,linsiz,nunits,units)
```

path-1	Input pathname.
len-1	Length of input pathname in characters, up to 80.
path-2	Output pathname.
len-2	Length of output pathname in characters, up to 80.
numkey	Number of pairs of keys (starting and ending columns). With binary keys, specifies number of 18.1 pairs of starting and ending bytes. Maximum is 64, default is 1.
nstart	Vector containing starting columns/bytes of keys. Each starting column must not be less than 1.
nend	Vector containing ending columns/bytes of keys. Each ending column must be no larger than linsiz.

npass Number of passes (returned).

nitem Number of items in output file (returned) — INTEGER*4.

nrev Vector containing sort order for each key:

0	Ascending
1	Descending

Default is 0 (ascending in Rev 19).

ispce Option to specify treatment of blanks:

0	Include blank lines in sort (default).
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, up to 80.

LOC(buffer) Obsolete — specify as 0. |18.1

msize Size (<65536) of common block for sort in words (INTEGER*2). It should be record size times maximum number of records expected. If nonzero, *msize* must be at least 1024 (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 is one segment (65536 words).

ntype Vector containing type of each key:

1	ASCII
2	16-bit integer
3	Single-precision real
4	Double-precision real
5	32-bit 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, lowercase sorts equal to uppercase
13	Unsigned integer

18.1

Default is all ASCII keys.

linsiz Maximum size of record in characters (bytes). Default is 32760.

nunits Obsolete.

units Obsolete.

Notes

1. Last four items are optional and may be omitted.

2. Files specified as merge files will be merged with the input file. Pathnames may be used for merge files.

▶ SRTF$S

Purpose

SRTF$S will sort input files (maximum 20) into a single output file. It is called by the previous two sorts.

Usage

```
CALL SRTF$S(inbuff,inlen,inunts,incnt,path2,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 pathnames in characters, up to 80.

inunts — Vector containing input file units (if open units are used).

incnt — Number of input files (up to 20).

path2 — Output file pathname.

len2 — Length of output pathname in characters, up to 80.

outunt — Output file unit (if an open unit is used).

numkey — Number of keys (pairs of starting and ending columns — starting and ending bytes if binary), up to 64. Default is 1. 18.1

nstart — Vector containing starting columns/bytes of keys. Each starting column number must be at least 1.

nend — Vector containing ending columns/bytes of keys. Each ending column must be no greater than the maximum input line size.

nrev — Vector containing sort order for each key:

0	Ascending (default)
1	Descending

ntype — Vector containing type of each key:

1	ASCII
2	16-bit integer
3	Single-precision real
4	Double-precision real
5	32-bit 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, lowercase sorts equal to uppercase.

18.1 13 Unsigned integer

Default is all ASCII keys.

erecode Return code.

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 record size in characters (bytes). Default is 32760. Required for sorting fixed-length records.

- inrec(3-5) Must be 0, 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 record size in characters (bytes).

- outrec(3-5) Must be 0, and are reserved for future use.

spcls Five-word array containing special options:

- spcls(1) Space option:
  - 0 Include blank lines in sort (default).
  - 1 Delete blank lines.

spcls(2) Collating sequence:

0	Default (ASCII at Rev. 19)
1	ASCII
2	EBCDIC

spcls(3) Tag/nontag option: 18.1

0	Default (tag sort at Rev. 19)
1	Tag sort
2	Nontag sort

spcls(4-5) Must be 0, and are reserved for future use.

msize Size of presort buffer in pages (units of 1024 words), not greater than 64. Note that the units used here are _pages_ which differ from the _words_ used by ASCSRT. Default is one segment (64 pages).

COOPERATING MERGE SUBROUTINES

To merge two or more sorted files with no special processing, use MRG1$S.

If postprocessing of the merged records is desired, the three merge subroutines in this chapter may also be used together in the following way. MRG1$S accepts specifications about the operation to be performed and the files and records to be used. The program should then call MRG2$S to get the merged records one at a time. Finally, the program calls MRG3$S to close units and delete temporary files opened by the other subroutines. 18.1

Many of the remarks about cooperating sort subroutines also apply to these merge routines. However, merging allows only output procedures. If MRG1$S is called with an output file (no output procedure), it calls MRG2$S and MRG3$S itself. If output is to a file, MRG2$S and MRG3$S should not be called.

▶ MRG1$S

Purpose

MRG1$S merges two to eleven previously sorted files into a single output file.

Usage

```
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 pathnames 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 pathname.

len2 — Length of output pathname in characters, up to 80.

outunt — Output file unit (if an open unit is used).

numkey — Number of pairs of keys (starting and ending columns — starting and ending bytes if binary), up to 64. Default is 1.

nstart — Vector containing starting columns/bytes of keys. Each starting column number must be at least 1.

nend — Vector containing ending columns/bytes of keys. Each ending column must be no greater than inrec(2).

nrev — Vector containing sort order for each key:

0 Ascending (default)

1 Descending

ntype — Vector containing type of each key:

1 ASCII

2 16-bit integer

3	Single-precision real
4	Double-precision real
5	32-bit 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, lowercase sorts equal to uppercase.
13	Unsigned integer 18.1

Default is all ASCII keys.

ercode Return code.

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 type specified in ntype.

inrec(2) Maximum input record size in characters (bytes). Required for sorting fixed-length records. Default is 32760.

inrec(3-5) Must be 0, 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 record size in characters (bytes).
- outrec(3-5) Must be 0, and are reserved for future use.

spcls — Five-word array containing:

- spcls(1) Space option:
  - 0 Include blank lines in sort (default).
  - 1 Delete blank lines.
- spcls(2) Collating sequence:
  - 0 Default (ASCII at Rev. 19)
  - 1 ASCII
  - 18.1 2 EBCDIC
- spcls(3-5) Must be 0, and are reserved for future use.

oproc — Output data destination (for use by MRG2$S):

- 0 Output file
- 1 Output procedure

▶ MRG2$S

Purpose

18.1 This subroutine is used only after MRG1$S has been called to set up the merge area, record and file specifications, and collating keys. MRG2$S returns the next merged record. MRG2$S should not be called for output files.

Usage

CALL MRG2$S (rtbuff, length)

rtbuff — Buffer containing next merged record (returned). Should be large enough to hold longest record merged.

length — Length of record (in characters) returned. Once all records have been returned, calls to MRG2$S return a length of 0.

▶ MRG3$S

Purpose

This subroutine is called after MRG1$S and MRG2$S. MRG3$S closes all units opened by the other merge routines. MRG3$S should not be called for output files.

Usage

18.1

CALL MRG3$S

COOPERATING SORT SUBROUTINES

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. These subroutines are available in V-mode only. All parameters are INTEGER*2.

A program can call the routines to work together in this way. SETU$$ sets up a table in which the sort is to be done, setting record size, record type, and other attributes. It also determines whether the records are to be read directly from the input files into the sort area, or whether they are to be accepted from an input procedure. It determines whether, after sorting, the records are to be sent directly to the output file or are to be postprocessed by an output procedure.

After calling SETU$S and giving it the necessary information, the program should call RLSE$S. If SETU$S has been told that there is to be a preprocessing input procedure, RLSE$S will take the record from its buffer. The input procedure is written by the user; it should call RLSE$S once for each record to be sorted. Otherwise, the arguments to RLSE$S will not be used, and RLSE$S will simply read the records from the input file(s) into the sort area.

Next, the program should call the sort procedure, CMBN$S, to do the actual sorting. Since SETU$$ should already have stored all information about record size, type, and collating sequence. CMBN$S accepts no parameters.

After CMBN$S, the program must call RTRN$S to take care of the sorted records. RTRN$S will either return records in the buffer specified in its parameter for postprocessing by an output procedure, or write them to the output file, according to the information already supplied to SETU$S.

18.1 Finally, the program calls CLNU$S to close files opened by RLSE$S and RTRN$S and to delete temporary sort files.

This combination of subroutines allows great flexibility in a sort operation, as the program that calls them can do a great deal of processing of the records before and after sorting. There is a tradeoff however; if you use input or output procedures, there is a procedure call for every single record, and the pre- or postprocessing itself adds time, so these routines will slow the sort.

An example of combined use of these subroutines is given below.

▶ SETU$S

Purpose

SETU$S checks the parameters supplied by the user and sets up all tables for the particular sort being defined.

Usage

```
CALL SETU$S  (inbuff,inlen,inunts,incnt,path2,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 pathnames in characters, up to 80.

inunts — Vector containing input file units (if open units are used).

incnt — Number of input files (up to 20).

path2 — Output file pathname.

len2 — Length of output pathname in characters, up to 80.

outunt — Output file unit (if an open unit is used).

numkey — Number of pairs of keys (starting and ending columns or starting and ending bytes if binary), up to 64. Default is 1. 18.1

nstart — Vector containing starting columns/bytes of keys (must be 1 or greater).

nend — Vector containing ending columns/bytes of keys (must be no greater than inrec(2)).

nrev — Vector containing sort order for each key:

0	Ascending (default)
1	Descending

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, lowercase sorts equal to uppercase.
13	Unsigned integer 18.1

Default is all ASCII keys.

ercode — Return code.

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 is 32760. Required for sorting fixed-length records.

inrec(3-5) Must be 0, 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 0, and are reserved for future use.

spcls Five-word array containing:

spcls(1) Space option:

0 Include blank lines in sort (default).

1 Delete blank lines.

spcls(2) Collating sequence:

18.1 0 Default (ASCII at Rev. 19)

1 ASCII

2 EBCDIC

spcls(3) Tag/nontag option:

0 Default (tag sort at Rev. 19)

1 Tag sort 18.1

2 Nontag sort

spcls(4-5) Must be 0, and are reserved for future use.

msize Size of common presort buffer in pages (units of 1024 words), no greater than 64. The size should be at least the product of the size of one record and the maximum number of records expected.

Default is one segment (64 pages).

iproc Input data source (used by RLSE$S):

0 Input file

1 Input procedure

oproc Output data destination (used by RTRN$S):

0 Output file

1 Output procedure

▶ RLSE$S

Purpose

RLSE$S transfers records from the buffer specified in the program or from an input file to the initial phase of the sort, according to the value of iproc in the SETU$S call.

Usage

```
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.

Discussion

If an input procedure is used, RLSE$S should be called once for each record released.

If an input file is used instead of an input procedure, RLSE$S should be called only once. If input is from a file, multiple calls to RLSE$S would result in multiple occurrences of each record when sorted.

Source records passed from an input procedure (when inrec(1) = 1 in the SETU$$ call) must end with a NEWLINE character (:212). Otherwise, the message "WARNING-LINE TRUNCATED" is issued and the last character is overwritten by a NEWLINE character. It is often easier to sort a text file as fixed-length records by reading them into the program with RDLIN$ rather than sorting them as source records.

▶ CMBN$S

Purpose

CMBN$S performs the internal sort. It uses the records provided by RLSE$S and the tables, collating sequence, and other information provided by SETU$S. If the sort cannot be done within allocated memory, CMBN$S merges the strings previously sorted.

Usage

```
CALL CMBN$S
```

▶ RTRN$S

Purpose

RTRN$S returns the records sorted by CMBN$S — to an output procedure or an output file, depending on the value of the oproc argument to SETU$S.

Usage

```
CALL RTRN$S(rtbuff,length)
```

rtbuff	Buffer containing next sorted record (returned). It 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 to return a record length of 0.

Discussion

If an output procedure is used, each call to RTRN$S obtains the next sorted record. The record is placed in rtbuff and must then be written to an output file, if it is to be saved.

If an output file is specified, RTRN$S is called only once.

If output is to a file, RTRN$S arguments are not used.

▶ CLNU$S

Purpose

CLNU$S closes all units opened by the sort routines and deletes any temporary files.

Usage

CALL CLNU$S

SAMPLE USER INPUT PROCEDURE

The following sample program demonstrates the use of an input procedure with the sort subroutines. This input procedure selects from INPUTFILE only those records beginning with AA for sorting.

```
OK, SLIST SAMPLE.FTN
C-----SAMPLE PROGRAM WHICH CALLS SORT
C-----TO DEMONSTRATE THE USE OF AN INPUT PROCEDURE BEFORE SORTING
C
C
$INSERT SYSCOM>KEYS.INS.FTN
$INSERT SYSCOM>ERRD.INS.FTN
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
C        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$NRTN,ERCODE,0,0,0,0)
C
C-----Initialize sort tables
C
      CALL SETU$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
C
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
C-----   (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$S (0,0)            /* send records to the output file
      CALL CLNU$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
```

This program may be compiled, loaded, and run with the following dialog:

```
OK, FTN SAMPLE -64V -DCLVAR
0000 ERRORS [<.MAIN.>FTN-REV18.4
OK, SEG -LOAD
[SEG rev 18.4
$ LO SAMPLE
$ LI VSRTLI
$ LI
LOAD COMPLETE
$ EXEC
```

The following listings show INPUTFILE and the sorted OUTPUTFILE.

```
OK, SLIST INPUTFILE
AA   EMPLOYEE1
BB   EMPLOYEE5
BB   EMPLOYEE3
CC   EMPLOYEE4
AA   EMPLOYEE2
AA   EMPLOYEE6
CC   EMPLOYEE7
AA   EMPLOYEE0


OK, SLIST OUTPUTFILE
AA   EMPLOYEE0
AA   EMPLOYEE1
AA   EMPLOYEE2
AA   EMPLOYEE6
OK,
```

SRTLIB (R-MODE) — SUBROUTINE DESCRIPTIONS

▶ SUBSRT

Purpose

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. Maximum keylength is 312 characters.

Usage

```
CALL SUBSRT(path-1,len-1,path-2,len-2,numkey,nstart,nend,npass,nitem)
```

path-1 Input pathname.

len-1 Length of input pathname in characters, up to 80.

path-2 Output pathname.

len-2 Length of output pathname in characters, up to 80.

numkey Number of keys (pairs of starting and ending columns — starting and ending bytes if binary). Maximum is 1, default is 1.

nstart Vector containing starting columns or bytes of keys.

nend Vector containing ending columns or bytes of keys.

npass Number of passes (returned).

nitem Number of items returned in output file (INTEGER*4).

▶ ASCS$$

Purpose

ASCS$$ is the R-mode subroutine that 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.

Usage

```
CALL ASCS$$  (path-1,len-1,path-2,len-2,numkey,nstart,nend,npass,
              nitem,nrev,ispce,mgcnt,mgbuff,len,LOC(buffer),msize,
              ntype,linsiz,nunits,units)
```

path-1 Input pathname.

len-1 Length of input pathname in characters.

path-2 Output pathname.

len-2 Length of output pathname in characters.

numkey Number of keys (pairs of starting and ending columns — starting and ending bytes if binary). Maximum is 20, default is 1.

nstart Vector containing starting columns or bytes of keys.

nend Vector containing ending columns or bytes of keys.

npass Number of passes (returned).

nitem Number of items returned in output file (INTEGER*4).

nrev Vector containing order for each key:

0 Ascending

1 Descending

ispce Whether to take blanks into account:

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 lengths of merge filenames in characters.

LOC(buffer) Location of presort buffer.

msize Size of presort buffer in words.

ntype Vector containing type of each key:

1 ASCII (default)

2	16-bit integer
3	Single-precision real
4	Double-precision real
5	32-bit integer

linsiz — Maximum size of record in characters -- optional. (Default is 508 characters.)

nunits — Number of file units available. (Optional — four will be used.)

units — Vector containing available file units (optional).

Discussion

The last four items are optional and may be omitted. Default value of ntype is ASCII.

Pathnames may not exceed 80 characters in length.

Files specified as merge files will be sorted and merged with the input file. Pathnames may be used for merge files, but only 10 merge files, each no more than 80 characters in length, may be used.

The 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.

MSORTS AND VMSORT - SUBROUTINE DESCRIPTIONS

These libraries contain several in-memory sort subroutines and a binary-search and table-building routine. MSORTS is the R-mode version, and VMSORT is the V-mode version. Each library contains the same subroutines.

The reference for most of the algorithms and timing studies is Donald Knuth, "Sorting and Searching," The Art of Computer Programming, vol. 3, Reading, MA: Addison-Wesley, 1973. It should be pointed out that the timing figures quoted are based upon Knuth's algorithms on his fictional machine (MIX). Since these routines are more general, 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.

The routines included in MSORTS and VMSORT 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

The binary search routine (BNSRCH) can be used either for table lookup in an ordered table or for building a sorted table.

All routines accept multiword entries and multiword keys located anywhere within the entry. The restrictions are that all entries are equal length and keywords 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 — Pointer to the first word of the table. (This is not a PL/I pointer.) For example, if the table is in an array TABLE(a,b), the parameter ptable = LOC (table). For routines in MSORTS, ptable is a full 16-bit pointer and can be in the upper 32K of memory. For VMSORT, ptable is a two-word pointer.

nentry — Number of table entries (not words) in the table. (That is, 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 per entry. nwds must be more than 0 . Obviously if nwds is greater than 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. nkwds must be greater than 0 and less than or equal to nwds. fword + nkwds - 1 must be no more than nwds. (In other words, the key field must be contained within an entry.)

npass Number of passes made (0 if error).

altbp Alternate return for bad parameters (used only with FORTRAN — use 0 for other languages).

RADXEX replaces the nkwds parameter with the following:

fbit First bit within fword of key. fbit must be greater than 0 and fword + (nbit+fbit - 2)/16 must be no more than nwds. (In other words, the key field must be contained within an entry.)

nbit Number of bits in key. 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, multiword integer. Therefore, the numbers 5, -1, 10, -3 would be sorted to -3, -1, 5, 10. RADXEX, since the key need not begin on a word boundary, treats the key as a single, unsigned multiword (or partial word) integer. Thus, the same four numbers would be sorted by RADXEX to 5, 10, -3, -1.

▶ BNSRCH

Purpose

BNSRCH sets up a binary table and performs a binary search.

Usage

```
CALL BNSRCH  (ptable, nentry, nwds, fword, nkwds, skey, fentry,
                index, opflag, altnf, altbp)
```

Most of these parameters are explained on the preceding page. The additional parameters are explained below.

skey Search key array (nkwds).

fentry Found entry array (nwds).

index Entry number of found entry.

opflag Operation key:

0 Locate.

1 Locate and delete.

2 Locate or insert.

3 Locate and update.

altnf Alternate return.

Discussion

Simple binary searching (opflag=0) tests each entry's key field 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 alternate return (altnf) is taken. If altnf is not specified, the normal return is taken, and the entry 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=0 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=0 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.

▶ BUBBLE

Purpose

Bubble sorting is a simple interchange sort.

Usage

```
CALL BUBBLE  (ptable, nentry, nwds, fword, nkwds, tarray, npass,
                altbp, incr)
```

Please read Parameters Common to More Than One Subroutine above.

incr   Used to sort nonadjacent entries. (See INSERT below.) Default is 1 (sort adjacent).

tarray   Must have nkwds words.

Discussion

Running Time: If N is the number of entries, 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

Purpose

Heap sort is based on a nonthreaded 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.

Usage

```
CALL HEAP (ptable, nentry, nwds, fword, nkwds, tarray, npass, altbp)
```

Please read Parameters Common to More Than One Subroutine above.

tarray   Must have nwds words.

Discussion

Running Time: If N is the number of entries, 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 QUICK.

▶ INSERT

Purpose

Straight insertion sorting is based upon "percolating" each element into its final position.

Usage

```
CALL INSERT (ptable, nentry, nwds, fword, nkwds, npass, altbp,
             incr)
```

Please read Parameters Common to More Than One Subroutine above.

incr   Used to sort nonadjacent entries.

Discussion

The incr parameter is used to sort nonadjacent entries. If, for example, incr = 3, every third entry will be included in the sort. The default is 1. For example, with incr equal to 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: Let N be the number of entries. 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.

▶ QUICK

Purpose

Quick is a partition exchange sort. QUICK is a variation of the basic quicksort called a median-of-three quicksort.

Usage

```
CALL QUICK  (ptable, nentry, nwds, fword, nkwds, tarray, npass,
             altbp)
```

Please read Parameters Common to More Than One Subroutine above.

tarray Must have nwds words.

Discussion

Running Time: If N is the number of entries, 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 should not be used on tables that are already well-ordered.

▶ RADXEX

Purpose

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.

Usage

```
CALL RADXEX  (ptable, nentry, nwds, fword, fbit, nbit, tarray,
              npass, altbp)
```

Please read Parameters Common to More Than One Subroutine above.

tarray Must have 2*nbit words; is used as partition stack.

Discussion

Running Time: If N is the number of entries, 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

Purpose

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 nonadjacent elements that are one INC apart. INC is then decreased on each pass. 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.

Usage

CALL SHELL (ptable, nentry, nwds, fword, nkwds, npass, altbp)

Please read Parameters Common to More Than One Subroutine above.

Discusion

Running Time: If N is the number of entries, 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.

PART V
Input/Output Library Subroutines

14
Introduction to IOCS

HOW TO USE PART V

IOCS (the Input/Output Control System) is a group of subroutines that perform input/output between the Prime computer and the disks, terminals, and other peripheral devices in the system. These subroutines have mostly been outdated by the ones in Chapters 9 and 10. Generally, these functions may be grouped into three levels:

Level 1 — Device-independent drivers are routines that read and write ASCII or binary and perform control functions such as opening a file.

Level 2 — Device-specific drivers 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 are routines that perform raw data transfers.

The chapters in Part V are organized in the following manner:

Chapter 14 — Device, unit, and argument definitions and tables for use with following chapters

Chapter 15 — How to change device assignments

Chapter 16 Device-independent driver subroutines (which call the device-dependent routines in the following chapters, depending on the device specified)

Chapter 17 Disk (non-file system) subroutines

Chapter 18 Subroutines for the user terminal and paper tape (Many subroutines may be used for both peripherals.)

Chapter 19 Subroutines for other peripheral devices (printers, plotters, card processors, and magnetic tape)

The level-1 device drivers are presented in Chapter 16. Routines of levels 2 and 3 are grouped in the following chapters by device type rather than by level of the subroutine.

Table 14-1 shows all IOCS routines discussed in Chapters 16-19. It shows the relationship of level-1 (device-independent) drivers to the others. Each column of this table represents an I/O function, and each horizontal row a certain physical device. All drivers in a single column are designed to be compatible in internal data format.

Tables 14-2 and 14-3 show the physical and logical device assignments, for use in changing device assignments as discussed in Chapter 15.

Figure 14-1 shows all the device-dependent drivers supported by Prime.

ARGUMENTS TO IOCS SUBROUTINES

The following argument names are used throughout Part V.

altrtn An INTEGER*2 variable assigned the value of a numeric label in the user's FORTRAN program, to be used as an alternate return from the subroutine in case of error. The label number should be preceded by a $. FORTRAN calls may omit the argument or give it the value of 0 if no alternate return is wanted. Other calling languages should omit the argument (not use 0).

buffer The name of a data area to or from which data is moved (integer array).

count The number of words to be transferred, or the length of a buffer or filename (INTEGER*2).

buffer-size The record size associated with the logical unit. Must be as large as the maximum record size.

logical-device Same as logical-unit below.

logical-unit The FORTRAN logical unit (Table 14-3).

name A filename.

physical-device The position in the device-type table (Table 14-2). A physical device is a device type such as magnetic tape or a user terminal.

physical-unit The sub-unit number of a physical device having more than one unit (Table 14-3). A physical unit designation distinguishes among the units of a physical device that has multiple units, such as a magnetic tape controller. For disk (the file system), the physical unit corresponds to the file unit (below). If the device has only one unit, its sub-unit number is 1. If it is a multiple-unit device such as disk or tape, sub-units 1 through 8 may be specified. (On disk, a sub-unit is actually processed as file 1-8.)

file unit The PRIMOS file-unit (funit) number from 0 through 127. (Users may assign 2 through 126.) File units are discussed in Chapter 9.

sub-unit The unit for multiunit devices (for disk, file unit number). This is the same as the physical unit (Table 14-3).

Table 14-1

Device-dependent Driver Selected by
Each Independent Driver According to Device

	Independent Drivers				
	RADSC	WRASC	RDBIN	WRBIN	CONTRL
Device	Dependent Drivers				
User terminal	I$AA01(6)	O$AA01(1)	I$BA01(2)	O$BA01(2)	C$A01(2)
18.1 Input command stream	I$AA12(1)				
Paper-tape reader	I$AP02(5)		I$BP02(2)		C$P02(5)
Paper-tape punch		O$AP02(5)		O$BP02(2)	
MPC card reader	I$AC03(3)	O$AC03(3)			
Serial line prtr.		O$AL04(3)			
9-track mag.tape	I$AM05(4)	O$AM05(4)	I$BM05(7)	O$BM05(7)	C$M05(4)
MCP line printer		O$AL06(4)			
PRIMOS file system compressed	I$AD07(1)	O$AD07(1)	I$BD07(1)	O$BD07(1)	SEARCH(1)
PRIMOS file system uncompr.	I$AD07(1)	O$AD08(1)	I$BD07(1)	O$BD07(1)	SEARCH(1)
Serial card rdr.	I$AC09(3)				
7-track mag.tape	I$AM10(4)	O$AM10(4)	I$BM10(7)	O$BM10(7)	C$M10(4)
7-track mag.tape BCD	I$AM11(7)	O$AM11(7)			C$M11(7)
9-track mag.tape EBCDIC	I$AM13(7)	O$AM13(7)			C$M13(7)
Versatec printer/plotter		O$AL14(3)			
MPC card processor	I$AC15(3)	O$AC15(3)			

* Numbers in parentheses refer to the following notes.

Notes to Table 14-1

1. Available in R-mode and V-mode. Listed in CONIOC (Chapter 15) and may be called directly or via the device-independent drivers.

2. Available in R-mode only. Listed in CONIOC (Chapter 15) and may be called directly or via the device-independent drivers.

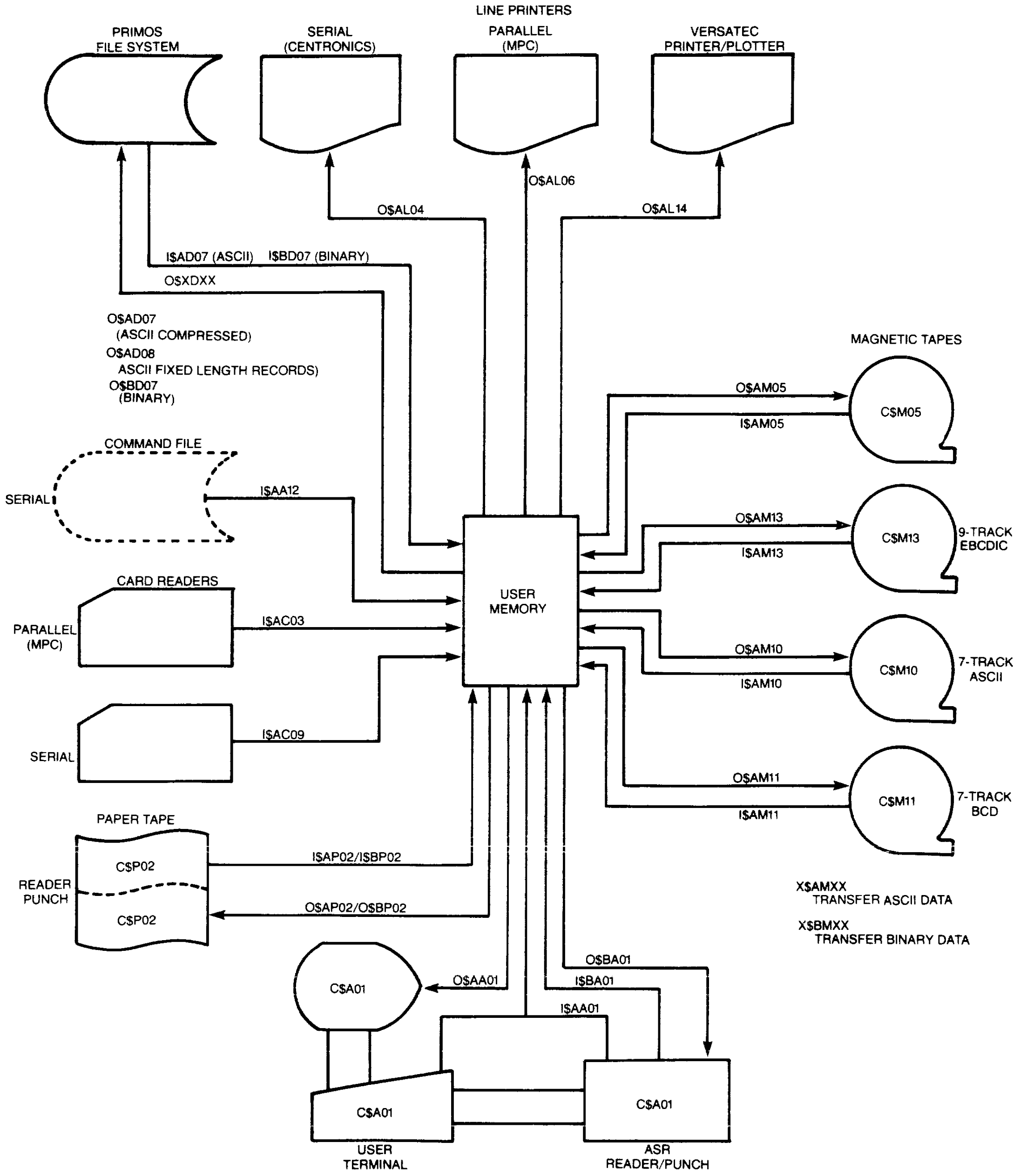
3. Available in R-mode only. Listed in FULCON but not CONIOC (Chapter 15). May not be called via the device-independent drivers, unless FULCON is assembled and loaded before the library is loaded.

4. Available in R-mode and V-mode. Listed in FULCON (Chapter 15). 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 nonshared V-mode library (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers unless FULCON is assembled and loaded before the library is loaded. See Chapter 15 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, routine is listed in CONIOC (Chapter 15) and may be called directly or via the device-independent drivers. For V-mode, routine is listed in FULCON (Chapter 15) 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 nonshared FORTRAN library (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers unless FULCON is assembled and loaded before the library is loaded. See Chapter 15 for a more complete discussion of IOCS table usage.

6. Available in R-mode and V-mode, but is not in CONIOC (Chapter 15) or FULCON. To call the routines via the device-independent drivers, the appropriate table must be modified, assembled, and loaded before the library is loaded. (See Chapter 15.) The routine may be called specifically without any special procedures.

7. Available in R-mode and V-mode. V-mode is listed in FULCON but not in CONIOC (Chapter 15). R-mode is not in CONIOC or FULCON. In V-mode, if the nonshared FORTRAN library (NPFTNLB) is loaded, the routine may not be called via the device-independent drivers unless FULCON 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.



Transfer of Data to and from High-speed User Memory
Figure 14-1

Table 14-2
Physical Device 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 Printer/Plotter

Table 14-3
Logical Devices, Physical Devices, and File Units

FORTRAN Default Logical Unit Number	Physical Device or Unit
1	User terminal
2	Paper-tape reader or punch
3	MPC card reader
4	Serial line printer (system option controller or SOC)
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
29	PRIMOS file unit 17
30	PRIMOS file unit 18
31	PRIMOS file unit 19
•	•
•	•
•	•
139	PRIMOS file unit 127
140	MPC printer 0 (AMLC)
141	MPC printer 1 (AMLC)

18.1

15
Device Assignment

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. Nonsharable devices may also be assigned on command level with the PRIMOS command ASSIGN. If a permanent device assignment is desired, the reader should go on to the next section of this chapter.

▶ ATTDEV

Purpose

ATTDEV attaches specified devices by associating logical-device with physical-device and associating the logical-device with a specific unit or file of the device.

Usage

```
CALL ATTDEV (logical-device, physical-device, physical-unit,
             buffer-size)
```

Note

For more discussion of arguments, see Chapter 14.

logical-device The device-independent logical I/O unit (Table 14-3). This number cannot be changed.

physical-device The position in the device-type tables (Table 14-2).

physical-unit The unit for multiunit devices (Table 14-3).

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 so that the logical unit has a current mapping.

Example

To reassign a card reader (logical unit 3) to physical device 2 (which has no sub-units) with the ability to read 80-column cards, enter:

```
CALL ATTDEV(3, 2, 0, 80)
```

Errors

If device is incorrect, ATTDEV returns the message:

```
ATTDEV BAD UNIT (unit-number)
```

PERMANENT DEVICE ASSIGNMENT

Users whose programs need to use devices other than the user terminal, the disks, or paper-tape reader or punch, or who wish to change the assignment of logical to physical devices must consult their System Administrator. The following discussion is an overview of the System Administrator's work.

To facilitate changes to device assignments, the tables used by IOCS (such as LUTBL and PUTBL) are in the following files on the master disk.

V-mode	VFINLIB>SOURCES>CONIOC.INS.PMA
R-mode	RFINLIB>IOCS>CONIOC.PMA

19

Ask your System Administrator how to locate the master disk on a multidisk system.

Note that the R-mode CONIOC.PMA in the RFINLIB 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.

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. There is a version of CONIOC that contains all the available IOCS drivers set up in the appropriate tables. This file is SOURCES>FULCON.INS.PMA in VFINLIB, or IOCS>FULCON.PMA in RFINLIB. The System Administrator can use FULCON as an example of how to set up CONIOC. The table entries that are not required can be set to 0.

The System Administrator may also change the default logical-to-physical-device association as given in Tables 14-2 and 14-3 by changing the IOCS tables RATBL, TBTBL, WATBL, and CNTBL in CONIOC. 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 14-2. PUTBL is the sub-unit table. The sub-unit table contains the individual unit or file numbers as required for multifile 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 unit 0, 1, 2, and 3 of 9-track magnetic tapes.

Modifying CONIOC to Change Device Assignment

Changing a device assignment is a System Administrator's responsibility and not a user function. The System Administrator may add or delete a device to any of the following tables.

RATBL	Read ASCII table.
RBTBL	Read binary table.
WATBL	Write ASCII table.
WBTBL	Write binary table.
CNTBL	Perform control function (endfile, rewind, etc.).

Input-only Devices

Input-only devices such as the card reader do not need WATBL and WBTBL entries. Furthermore, an ASCII-only device (such as a line printer) does not need RBTBL and WBTBL entries.

Order of Entries

The order of entries in the above-mentioned tables corresponds to physical-device numbers defined in Table 14-2.

R-mode Procedures

1 Attach to RFTNLIB>IOCS of Master disk A.

2 Edit the appropriate tables within CONIOC.PMA.

3 Replace the 0 with the corresponding subroutine name for the desired device.

4 Rebuild the RFTNLIB library. (See below.)

19

V-mode Procedures

1 Attach to VFTNLIB>SOURCES of Master Disk A.

2 Edit the appropriate tables within the CONIOC.INS.PMA.

3 Replace the word NULLDEVICE with the appropriate device subroutine name.

4 Rebuild the VFTNLIB Library. (See below.)

How to Rebuild the FORTRAN Library after Modifying CONIOC

R-mode Procedures: The R-mode FORTRAN library must be rebuilt after CONIOC has been modified:

1 Attach to RFINLIB on Master Disk A.

2 Run RFINLIB.BUILD.CPL.

3 Run INSTALL_FINLIB.CPL.

4 Share the new library (a System Administrator procedure). 19

V-mode Procedures: The V-mode FORTRAN library must be rebuilt after CONIOC has been modified:

1 Attach to UFD = VFINLIB on Master Disk A.

2 Run VFINLIB.BUILD.CPL.

3 Share the new library (a System Administrator procedure).

16
Device-independent Drivers

This chapter presents the subroutines listed in the top (horizontal) row of Table 14-1. They have the following functions:

Routine	Function
WRASC	Write ASCII
RDASC	Read ASCII
WRBIN	Write binary
RDBIN	Read binary
CONTRL	Other control functions

To maintain device independence, all data transfers can be accomplished through these five device-independent drivers in IOCS. These device-independent or first-level drivers route the I/O request to one of the device-dependent drivers, as shown in Table 14-1 and Figure 14-1. The device-dependent drivers are presented in the following chapters (17 through 19). Each column of Table 14-1 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.

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 NEWLINE (octal 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.

ARGUMENTS FOR DEVICE-INDEPENDENT DRIVERS

The device-independent drivers all have the same arguments. The arguments are defined in Chapter 14.

DESCRIPTION OF SUBROUTINES

▶ WRASC

Purpose

WRASC writes ASCII characters to any output device.

Usage

```
CALL WRASC (logical-device,buffer,count,altrtn)
```

Discussion

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.

▶ RDASC

Purpose

RDASC reads ASCII characters from any input device.

Usage

```
CALL RDASC (logical-device,buffer,count,altrtn)
```

Discussion

One record is brought into memory. Buffer is always filled with count ASCII characters, two per word. If the record is 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.

WRBIN

Purpose

WRBIN writes binary data to any output device.

Usage

```
CALL WRBIN (logical-device,buffer,count,altrtn)
```

Discussion

The number of words specified by count are written from buffer to the specific output device. The format of the data is device-dependent.

RDBIN

Purpose

RDBIN reads binary input from any input device.

Usage

```
CALL RDBIN (logical-device,buffer,count,altrtn)
```

Discussion

A record is read into memory. Count is the maximum number of words that 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.

CONTRL

Note

This subroutine is obsolete, and has been replaced with SRCH$$ (Chapter 9).

Purpose

Certain nondata transfer functions, such as opening a PRIMOS file for reading, are provided by use of the CONTRL subroutine.

Usage

CALL CONTRL (key, name, logical-device, altrtn)

key — A numeric option code that may have the following values:

1 Open for reading.

2 Open for writing.

3 Open for read/write.

4 Close.

5 Delete file.

6 Move forward one 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, and must be read by a user-written PMA subroutine.

-1 Write file mark (MT only).

-2 Backspace one record (MT only).

-3 Backspace one file mark (MT only).

-4 Rewind to beginning of tape (MT only).

Note

For calls to disk files, key may have many other values. See SRCH$$. Keys other than 1-4 are not device-independent.

name	Filename (0 if none).
logical-device	See Chapter 14.
altrtn	See Chapter 14.

Discussion

Functions not applicable to a particular device are ignored; therefore, functions can be requested in a device-independent way. See Table 16-1 for operation effects.

Table 16-1
List of Keys and Operating Effects for CONTRL

Key	Terminal (C$A01)	Paper-tape Reader/Punch (C$P02)	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 one 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.

17
Disk Subroutines

This chapter defines the subroutines for non-file-system disk I/O operations. The first set is a subset of the device-dependent drivers listed in Table 14-1. They comprise the drivers listed in the rows for the PRIMOS file system, except for SRCH$$, which is presented in Chapter 9. Most users will find that other subroutines, in Chapters 9 and 12, will perform I/O functions faster and with more options than these drivers.

The second section of the chapter lists some obsolete disk subroutines: D$INIT, WRECL, and RRECL.

These are the subroutines presented in this chapter:

Routine	Meaning
O$AD07	Write ASCII to disk.
I$AD07	Read ASCII from disk.
O$BD07	Write binary to disk.
I$BD07	Read binary from disk.
O$AD08	Write ASCII to disk (fixed-length records).
D$INIT	Initialize disk (obsolete).

RRECL	Read one disk record (obsolete).
WRECL	Write one disk record (obsolete).

ARGUMENTS

The arguments for these subroutines are defined in Chapter 14.

DRIVER SUBROUTINES

▶ O$AD07

Note

> This subroutine is obsolete, and has been replaced with WTLIN$ (Chapter 9).

Purpose

O$AD07 writes ASCII from buffer onto a disk file open on file-unit.

Usage

```
CALL O$AD07 (file-unit, buffer, count, altrtn)
```

For an explanation of arguments, see Chapter 14.

Discussion

Information is written on the disk in compressed ASCII format. Multiple blank characters are replaced with the character DC1 (221 octal) followed by a word count. Trailing blanks are removed and the end of record indicated by the NEWLINE character, or NEWLINE followed by null.

▶ I$AD07

Purpose

I$AD07 reads information from the disk file open on file-unit, in compressed ASCII format.

Usage

CALL I$AD07 (file-unit, buffer, count, altrtn)

For an explanation of arguments, see Chapter 14.

▶ O$BD07

Purpose

O$BD07 writes binary information to the file open on file-unit.

Usage

CALL O$BD07 (file-unit, buffer, count, altrtn)

For an explanation of arguments, see Chapter 14.

▶ I$BD07

Purpose

I$BD07 reads binary information from the file open on file-unit.

Usage

CALL I$BD07 (file-unit, buffer, count, altrtn)

For an explanation of arguments, see Chapter 14.

▶ O$AD08

Purpose

O$AD08 writes ASCII from buffer onto the disk file open on file-unit.

Usage

CALL O$AD08 (file-unit, buffer, count, altrtn)

For an explanation of arguments, see Chapter 14.

Discussion

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.

OBSOLETE DISK SUBROUTINES

These subroutines are not in FINLIB. They were intended for use by the System Administrator.

▶ D$INIT

Purpose

The D$INIT routine is called to initialize disk devices.

Usage

CALL D$INIT (pdisk)

pdisk	The physical disk number to be initialized. (See RRECL below.)

Discussion

D$INIT initializes the disk controller and performs a seek to cylinder 0 on pdisk. D$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 was intended by use only by outdated system utilities.

▶ RRECL

Purpose

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 was intended for use only by now outdated system utilities such as FIXRAT, MAKE, and the old disk COPY.

Usage

```
CALL RRECL (LOC(buffer), length, option-word, ra, pdisk, altrtn)
```

buffer — An array into which length words from record ra will be transferred.

length — The number of words to be transferred.

option-word — A 16-bit word with the following options:

- 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 ra. (This feature may be used only if RRECL is running under PRIMOS II, 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.
- Bits 5-8 — Reserved.
- Bits 9-16 — Must be set on (1).

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 of PRIMOS.

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. (See Chapter 14.)

Discussion

If an error is encountered and control goes to altrtn, ERRVEC (Appendix E) is set as follows:

Code	Message	Meaning
ERRVEC(1) = WB	On supervisor terminal: 10 times	Disk hardware
ERRVEC(2) = 0	DISK RD ERROR pdisk ra status	WRITE PROTECT error
	On user terminal: UNRECOVERED ERROR	

ERRVEC(1) = WB	On user terminal: 10 times	Current record address error
ERRVEC(2) = CR	DISK RD ERROR *pdisk* *ra* status followed by UNRECOVERED ERROR	

See the *System Administrator's Guide* for a description of status error codes.

Notes

Length must be between 0 and 448 unless *pdisk* is a storage module, in which case *length* must be between 0 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 the *option-word*'s bit 2 to 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 0 and a reread of the record. If 10 errors occur, the message UNRECOVERED ERROR is typed to the user or *altrtn* is taken.

▶ WRECL

Purpose

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 *option-word*, which have no meaning on write. For a call to write a record on the diskette, the buffer *length* must be 448 words.

D$INIT must be called before a call to WRECL.

Usage

CALL WRECL (LOC(buffer), length, option-word, 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   option-word   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 RRECL.)

18 User Terminal and Paper-Tape Subroutines

OVERVIEW

This chapter defines subroutines used to transfer data to and from a user terminal or card reader/punch (ASR). Some are a subset of the device-dependent IOCS drivers shown in Table 14-1, in the rows for the user terminal and for paper tape. Other subroutines in this chapter are of general use for these devices. They are listed elsewhere, and referenced here for completeness of the user-terminal and paper-tape chapter.

The subroutines in this chapter are listed in Table 18-1.

LIST OF SUBROUTINES

▶ BREAK

Purpose

BREAK inhibits or enables CONTROL-P.

Usage

For the calling sequence and discussion, see Chapter 10.

Table 18-1
Subroutines for User Terminal and Paper Tape

Device	Routine	Function
User terminal	BREAK	Inhibits or enables CONTROL-P.
	C1IN	Gets next character from terminal or command file.
	CNIN$	Moves characters from terminal or command file to memory.
	COMANL	Reads a line of text from the terminal or from a command file.
	ERKL$$	Reads or sets erase and kill characters.
	TNOU	Outputs count characters to the user terminal followed by the LINEFEED and carriage return.
	TNOUA	Outputs count characters to the user terminal.
	TOVFD$	Outputs the 16-bit integer num to the terminal.
	T1IB	Reads one character from the user terminal into Register A.
	T1OB	Writes one character from Register A to the user terminal.
	T1IN	Reads one character from the user terminal.
	T1OU	Outputs char to the user terminal. The data type must be a 16-bit integer in F77.
	TIDEC	Inputs decimal number.
	TIOCT	Inputs an octal number.
	TIHEX	Inputs a hexadecimal number.
	TODEC	Outputs a six-character signed decimal number.
	TOOCT	Outputs a six-character unsigned octal number.

Table 18-1 (continued)
Subroutines for User Terminal and Paper Tape

Device	Routine	Function
	TOHEX	Outputs a four-character unsigned hexadecimal number.
	TONL	Outputs carriage return and LINE-FEED.
	C$A01	Controls functions for user terminal.
User terminal or ASR punch	O$AA01	Outputs ASCII to the user terminal or ASR punch.
Keyboard or ASR reader	I$AA01	Inputs ASCII from terminal or ASR reader.
	I$AA12	Performs the same function as I$AA01 but also allows the input to be from a cominput file.
Paper tape	I$AP02	Inputs ASCII from the high-speed paper-tape reader.
	P1IB	Inputs one character from the high-speed paper-tape reader to Register A.
	O$BP02	Outputs binary data to the high-speed paper-tape punch.
	P1OB	Outputs one character to the high-speed paper-tape punch from Register A.
	P1OU	Outputs one character to the high-speed high-speed paper-tape punch.
	P1IN	Inputs one character from paper tape, sets high-order bit, ignores line feeds, sends a line feed when carriage return is read.
	C$P02	Controls functions for paper tape.

▶ C$A01

Purpose

C$A01 provides control functions for the user terminal.

Usage

CALL C$A01 (key, name, physical-unit [, altrtn])

Arguments are explained in Chapter 14; key is in Table 16-1.

▶ C$P02

Purpose

C$P02 provides control functions for paper tape.

Usage

CALL C$P02 (key, name, physical-unit [,altrtn])

Arguments are explained in Chapter 14; key is in Table 16-1.

▶ C1IN

Purpose

C1IN gets the next character from the terminal or command file.

Usage

For the calling sequence and discussion, see Chapter 10.

▶ CNIN$

Purpose

CNIN$ moves characters from the terminal or a command file to memory.

Usage

For the calling sequence and a discussion, see Chapter 10.

▶ COMANL

Purpose

COMANL reads a line of text from the terminal or from a command file.

Usage

For the calling sequence and a discussion, see Chapter 10.

▶ ERKL$$

Purpose

ERKL$$ reads or sets the erase and KILL characters.

Usage

For the calling sequence and a discussion, see Chapter 10.

▶ I$AA01

Purpose

I$AA01 reads ASCII from the terminal or ASR reader.

Usage

```
CALL I$AA01 (sub-unit, buffer, count [,altrtn])
```

For a discussion of arguments, see Chapter 14.

Discussion

The kill and erase characters (question mark and quote mark by default) 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 EXT (octal 203) indicated the end of file and is used for reading tapes through the user terminal.

Note that I$AA01 is not the entry for the user terminal in the Prime-supplied CONIOC (Chapter 15). Put I$AA01 in the table as explained in Chapter 15 to read paper tapes with user programs. The editor should be used to read in the tape, and then the user may read the file from disk.

▶ I$AA12

Purpose

I$AA12 performs the same function as I$AA01 but also allows the input from a cominput file.

18.1

Usage

CALL I$AA12 (sub-unit, buffer, count[, altrtn])

For a discussion of arguments, see Chapter 14.

▶ I$AP02

Purpose

I$AP02 reads ASCII from the high-speed paper-tape reader.

Usage

CALL I$AP02 (sub-unit, buffer, count[, altrtn])

Discussion

The KILL and ERASE characters (question mark and double quote by default) modify the input. NUL, DEL, DLE, DC2, DC3, and DC4 are ignored. The character ETX (octal 203) indicates end of file.

▶ O$AA01

Purpose

O$AA01 outputs ASCII to the user terminal or ASR punch.

Usage

CALL O$AA01 (sub-unit, buffer, count[, altrtn])

For a discussion of arguments, see Chapter 14.

Discussion

This subroutine calls the driver TNOU.

▶ O$BP02

Purpose

O$BP02 writes binary data to the high-speed paper-tape punch.

Usage

CALL O$BP02 (sub-unit, buffer, count[, altrtn])

For a discussion of arguments, see Chapter 14.

Discussion

The format of the paper-tape output can be found in a listing of this driver. Ask your System Administrator how to obtain a copy of the listing.

▶ P1IB

Purpose

P1IB reads one character from the high-speed paper-tape reader to Register A.

Usage

CALL P1IB

This subroutine has no arguments; the calling program must have access to Register A.

▶ P1IN

Purpose

P1IN reads one character from paper tape.

Usage

CALL P1IN (char)

Discussion

The subroutine sets the high-order bit, ignores line feeds, and sends a line feed when a carriage return is read.

▶ P1OB

Purpose

P1OB writes one character to the high-speed paper-tape punch from Register A.

Usage

CALL P1OB

This subroutine has no arguments; the calling program must have access to Register A.

▶ P1OU

Purpose

P1OU writes one character to the high-speed paper-tape punch.

Usage

CALL P1OU (char)

Zero the high-order bit before punching. No special action is taken on carriage returns or line feeds.

▶ T1IB

Purpose

T1IB reads one character from the user terminal into Register A.

Usage

CALL T1IB

This subroutine has no arguments; the calling program must have access to Register A.

▶ T1OB

Purpose

T1OB writes one character from Register A to the user terminal.

Usage

CALL T1OB

This subroutine has no arguments; the calling program must have access to Register A.

▶ T1IN

Purpose

T1IN reads one character from the user terminal.

Usage

CALL T1IN (char)

Discussion

If a carriage return is read, a NEWLINE is output and char is set to NEWLINE. If a NEWLINE is read, a carriage return is output and char is set to NEWLINE.

If .XOF. is read, a carriage return and NEWLINE are expected to follow. T1IN ignores the .XOF., reads the carriage return and line feed, then sets char to NEWLINE. The .XOF. characters are expected on paper tape.

▶ T1OU

Purpose

T1OU writes a character to the user terminal.

Usage

CALL T1OU (char)

The data type of char must be a 16-bit integer in FORTRAN IV or FORTRAN 77. If char is NEWLINE, the characters carriage return and NEWLINE are output to the user terminal.

▶ TIDEC

Purpose

TIDEC reads terminal input as a decimal number.

Usage

CALL TIDEC (variable)

Discussion

The number may be preceded by a minus to indicate that it is negative, but must not be preceded by a plus sign. Numbers may be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is improper, and more input will then be accepted. A space or carriage return will then be accepted as a 0.

▶ TIHEX

Purpose

TIHEX reads terminal input as a hexadecimal number.

Usage

CALL TIHEX (variable)

Discussion

The number may be preceded by a minus to indicate that it is negative, but must not be preceded by a plus sign. Numbers may be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is improper, and more input will then be accepted. A space or carriage return will then be accepted as a 0.

▶ TIOCT

Purpose

TIOCT reads terminal input as an octal number.

Usage

CALL TIOCT (variable)

Discussion

The number may be preceded by a minus to indicate that it is negative, but must not be preceded by a plus sign. Numbers may be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is improper, and more input will then be accepted. A space or carriage return will then be accepted as a 0.

▶ TNOU

Purpose

TNOU writes count characters to the user terminal followed by a LINEFEED and carriage return.

Usage

CALL TNOU (buffer, count)

Buffer is expected to contain two characters per word.

This subroutine is especially useful for the transfer of nonprinting characters.

▶ TNOUA

Purpose

TNOUA writes count characters to the user terminal.

Usage

CALL TNOUA (buffer, count)

Discussion

This subroutine is especially useful for transfer of nonprinting characters.

Example

For an example, see the first sample program of the COBOL chapter.

▶ TODEC

Purpose

TODEC outputs a six-character signed decimal number.

Usage

```
CALL TODEC (variable)
```

▶ TOHEX

Purpose

TOHEX outputs a four-character unsigned hexadecimal number.

Usage

```
CALL TOHEX (variable)
```

▶ TOOCT

Purpose

TOOCT outputs a six-character unsigned octal number.

Usage

```
CALL TOOCT (variable)
```

▶ TONL

Purpose

TONL outputs a carriage return and line feed.

Usage

```
CALL TONL
```

▶ TOVFD$

Purpose

TOVFD$ writes a 16-bit integer to the terminal.

Usage

CALL TOVFD$ (number)

Discussion

This subroutine writes number, which should be a 16-bit integer, to the terminal without any spaces (for example, 123 or -17).

19
Other Peripheral Devices

This chapter, describes subroutines that control line printers, printers/plotters, card readers, and magnetic tapes. These subroutines are used for both formatted and raw data. Not all are in IOCS. They are listed in Table 19-1.

LINE PRINTER SUBROUTINES

IOCS contains subroutines to control three types of line printers. They are: O$AL04 to print on a Centronics Line Printer connected to the system option controller (SOC); O$AL06 to print on a parallel-interface line printer connected to the MPC Line Printer Controller; and O$AL14 to print on a Versatec Printer/Plotter connected to a Versatec-SOC Controller. This section also includes SPOOL$ for queuing files to be printed, and T$LMPC to move data to the MPC line printer.

Table 19-1
Peripheral-handling Subroutines

Line Printers	
O$AL04	Centronics LP.
O$AL06	Parallel interface to line printer (MPC).
O$AL14	Versatec printer.
T$LMPC	Move data to MPC line printer.
SPOOL$	Insert a file in spooler queue.
Printer/Plotter	
T$VG	Versatec.
O$AL14	Versatec.
Card Reader/Punch	
I$AC03	Input from parallel card reader.
I$AC09	Input from serial card reader.
I$AC15	Read and print card from parallel interface reader.
T$CMPC	Input from MPC card reader.
O$AC03	Parallel interface to card punch.
O$AC15	Parallel interface to card punch and print on card.
T$PMPC	Raw data mover.
Magnetic Tape	
C$M05	Control functions for 9-track ASCII/binary.
C$M10	Control functions for 7-track ASCII/binary.
C$M11	Control functions for 7-track EBCDIC.
C$M13	Control functions for 9-track EBCDIC.
O$AM05	Write ASCII to 9-track.
O$AM10	Write ASCII to 7-track.
I$AM05	Read ASCII from 9-track.
I$AM10	Read ASCII from 7-track.
O$BM05	Write binary to 9-track.
O$BM10	Write binary to 7-track.
I$BM05	Read binary from 9-track.
I$BM10	Read binary from 7-track.
O$AM11	Write BCD to 7-track.
O$AM13	Write EBCDIC to 9-track.
I$AM11	Read BCD from 7-track.
I$AM13	Read EBCDIC from 9-track.
T$MT	Raw data mover.

▶ O$ALxx

Purpose

These subroutines provide an interface to the line printers. O$AL14 is discussed separately below.

Usage

```
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.

Discussion

For more information on arguments, see Chapter 14.

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:

- forms control
- Header line and pagination control
- No-control

O$ALxx checks the first character in the data buffer for a .SOM. or start-of-message character (ASCII :001). This character signifies a change in the control mode. If the first character in the buffer is not .SOM., the line is printed according to the current control mode. The default mode is forms control.

Forms Control Mode

The first character in the buffer is not printed; instead, it is used for forms control. The character interpretations are as follows:

Character	Interpretation
0	Skip a line.
1	Eject to top of next page.
+	Overprint last line (AL06 only).
Any character other than 0, 1, +	No action.

Header Line and Pagination Control Mode

In header line and pagination 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.

For O$AL06 and O$AL14, enter pagination mode with a first word of :000001 in buffer. In pagination mode with O$AL04, a form feed (octal 14 or 214) may be anywhere in the buffer line. All characters preceding the form feed are printed, and all characters after it are ignored. With O$AL04, the form feed must be in column 1 or 3.

No-control Mode

In No-control mode, no actions are taken by O$ALxx. A line containing an ASCII formfeed character (FF, :214) 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. LINEFEED (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
000	Enter no-control mode.
001	Enter control mode.
036	New header line - DO NOT reset page count.
037	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.

▶ O$AL14

Purpose

O$AL14 provides the IOCS interface to the Versatec printer.

Usage

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. (See Chapter 14.)

Discussion

The action taken by O$AL14 depends upon the data in the buffer and the current vertical control mode (first character of buffer).

O$AL14 has three vertical control modes:

1. Forms control
2. Header line and paginate control
3. No-control

The default mode is forms control. O$AL14 checks the first character in the data buffer for a .SOM. (ASCII :001). This character signifies a change in the control mode. If the first character is not a .SOM., the line is printed according to the current control mode. Mode descriptions follow.

Forms Control: In this mode, the first character in a buffer is never printed but is used for forms control. The character interpretations are:

0	Skip one line.
1	Eject to top of next page.
+	Print over last line (if printer model allows).
Other	No action.

Header Line and Pagination: 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 a .SOM. will cause an internal change by O$AL14. The change is determined by the character following the .SOM.:

000 Enter no-control mode.

001 Enter control mode.

036 New header line but do not reset page count.

037 Enter new page size specified by the 16-bit number contained in the next computer word.

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.

Load information: This subroutine calls T$VG.

▶ T$LMPC

Purpose

The T$LMPC routine is the raw data mover that moves information from the user to one line on the MPC line printer.

The user normally prints lines under program control using either FORTRAN WRITE statements or a call to O$AL06, which in turn calls T$LMPC. However, it is possible to call T$LMPC directly.

Usage

CALL T$LMPC (logical-unit, LOC(buffer), count, instr, status)

logical-unit — Line printer unit.

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:

Octal Value	Condition
200	Online
100	Not busy

Discussion

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 one 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$

Purpose

A user program can insert a file into the spool directory by calling the SPOOL$ subroutine.

Usage

```
CALL SPOOL$ (key, name, namlen, info, buffer, buflen, code)
```

key — User option:

- 1 — Copy named file into queue.
- 2 — Open file on unit info(2) for writing.

name — File to be copied (if key=1), or name to appear on header page (if key=2).

namlen — Length of name, in characters (1-32).

info — Information array, 12 to 29 words, as follows:

- 1 — Reserved after Rev. 17.
- 2 — Temp file unit 2 (may range from 1-126 for Rev. 17 and above).
- 3 — Print option word. (See below.)
- 4-6 — Form type (6 ASCII characters). (Equivalent to -FORM on PRIMOS command line.)
- 7 — Plot raster scan size (plot only). This represents number of words/raster scan.
- 8-10 — Spool filename (returned).
- 11 — Deferred print time (valid only if defer bit specified in option word) - an integer specifying minutes after midnight. (Equivalent to -DEFER in PRIMOS command line.)
- 12 — File size, returned if key is 1.
- 13-20 — (Optional) Logical destination name — must be blank-padded (equivalent to -AT

on command line). If these words are used, bit 10 of word 3 must be set to 1.

21-28 (Optional) Substitute filename to be used — must be blank-padded (equivalent to -AS on command line). If these words are used, bit 11 of word 3 must be set to 1.

29 (Optional) Number of copies (equivalent to -COPIES on command line). If this word is used, bit 12 of word 3 must be set to 1.

buffer Scratch buffer - this is used to set up control info and to copy the file to the spool queue if key is 1. It must be at least 40 words long. Copy time is inversely proportional to buffer size. Nominal size is between 300 and 2000 words.

buflen Length of buffer.

code Return code (nonzero for file system error).

Word 3 of the information array (print option word) is defined as follows:

Bit	Meaning If Set to 1
1	Format control. (Column 1 contains carriage control information.)
2	Expand compressed listing.
3	Generate line numbers 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 — Not used after Rev. 17.
10	If 1, use the logical destination name specified in info(13-20).

11 If 1, use the substitute filename specified in info(21-28).

12 If 1, spool the number of copies specified in info(29).

13-16 Reserved.

PRINTER/PLOTTERS

The printer/plotter subroutines are used to drive and control the Versatec printer/plotter.

▶ T$VG

Purpose

T$VG moves raw data from a buffer and prints the data on the Versatec printer via a controller designed for use with the Versatec printer/plotter.

Usage

```
CALL T$VG (physical-unit,LOC(buffer),nwds,instruction,status)
```

physical-unit Currently always 0, since the controller supports only one device.

LOC(buffer) Address of user's buffer.

nwds The number of words in the buffer. The maximum is 512.

instruction A number from 0 to 10 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) is as follows:

Bit	Meaning
1	Always 0.
2	If=1, then paper is low.
3	If=0, then printer/plotter is ready. If=1, printer/plotter is not ready.
4	If=0, printer/plotter is online otherwise, printer/plotter is offline.
5-16	Always 0.

Printer/Plotter Instructions

Instructions to the printer/plotter are specified in the instruction field of the calling sequence. They are a number from 1 to 10 interpreted as follows:

0 Return printer/plotter status in status(2). The contents of the status vector, status, are described in the calling sequence description. T$VG waits until the output buffer is empty or until there is a timeout before returning status.

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 only — see below.)

7 Make a plot, using the contents of buffer. (Plot mode only — see below.)

8 Simultaneous print/plot PRINT. (SPP mode only — see below.)

9 Simultaneous print/plot PLOT. (SPP mode only — see below.)

10 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 0. If there is room for the number of words specified by nwds, set status(2) to a nonzero 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	LINEFEED. The transmission of a LINEFEED 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 (leftmost) bit, and bit 16 of memory word is the least significant (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 Versatec 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 19-2 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 19-2
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

CARD PROCESSING SUBROUTINES

Card-reader subroutines drive and control serial and parallel interface card readers.

Card Reading Operation

The user must insert the card deck in the card reader and give the command:

```
ASSIGN  CRn
```

n =0 or 1 for the device sub-unit number

The user then fills the input buffer from the card reader by calling subroutines T$CMPC, T$PMPC (operating system library), or I$AC03, I$AC15 (FORTRAN library).

The user may issue a status request call to check if the input buffer is empty. If the buffer is empty, the online status bit (bit 9 in the status word) is reset.

Note

Under PRIMOS II, the card reader is never offline.

▶ I$AC03

Purpose

Reads ASCII input from the parallel interface card reader.

Usage

```
CALL I$AC03 (physical-unit, buffer, word-count, altrtn)
```

physical-unit Device to or from which data is to be moved:

0 CR0, 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. (See Chapter 14.)

Discussion

Card Format: Cards are expected to be in 029 format. '026' cards may be read by preceding the deck by a card containing '$6' in columns 1 and 2. The conversion done for '026' cards is shown below.

Card Code (026 Symbol)	Converted to (Character)
#	=
%	(
<	)
@	'
&	+

The driver can be switched back to '029' format by '$9' in columns 1 and 2.

Load Information: This subroutine calls T$CMPC.

▶ I$AC09

Purpose

The subroutine I$AC09 reads ASCII input from a serial interface card reader.

Usage

CALL I$AC09 (unit, buffer-name, word-count, altrtn)

Discussion

I$AC09 translates card codes to characters in memory as follows:

Card Code (026 Symbol)	Converted to (Character)
#	=
%	(
<	)
+	&
&	+
@	'

Card codes read are either 026 or 029. The last card in the deck is .Q..

Errors: The ERRVEC(3) may have the following octal values. (See Appendix E for a discussion of ERRVEC.) Combinations are possible.

200	Online
40	Illegal ASCII
20	DMX overrun
4	Hopper empty
2	Motion check
1	Read check

Load Information: I$AC09 calls F$AT to fetch the arguments.

▶ I$AC15

Purpose

Reads and interprets (prints) a card from a parallel interface card reader.

Usage

CALL I$AC15(physical-unit, buffer, word-count, altrtn)

physical-unit   Card-reader sub-unit:

0	CR0, first controller
1	CR1, second controller

buffer   Data name into which card is to be read.

word-count   Number of words to be read.

altrtn   Alternate return in case of error. (See Chapter 14.)

Load Information

This subroutine calls T$PMPC.

▶ T$CMPC

Purpose

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.

T$CMPC is called by the IOCS card-reader driver I$AC03. The user normally reads cards under program control using either FORTRAN READ statements or a call to I$AC03. However, it is possible to call T$CMPC directly.

Usage

```
CALL T$CMPC(physical-unit, LOC(buffer), word-count, instr, status)
```

physical-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.

instr The instruction required to be sent to the card reader. Valid instructions are:

Instruction	Meaning
100000 (octal)	Return status.
40000 (octal)	Read card in ASCII format.
60000 (octal)	Read card in binary format.
100001 (octal)	Return status of hardware.

status A three-word vector:

status(1) Not used.

status(2) Card-reader status: If status is explicitly requested by instr (:100000), this word returns a value indicating the state of buffer (not of the hardware). Otherwise the status bits returned are defined as follows:

Octal Value	Condition
200	Online
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$CMPC (0, LOC(CARDS), 40, :40000, STATUS)
    60        CALL O$....  /*SAVE CONTENTS OF CARDS
    70        CONTINUE
```

The above example reads an 80-character card of ASCII data and places the contents in CARDS.

▶ O$AC03

Purpose

O$AC03 punches output to the parallel interface card punch.

Usage

CALL O$AC03(physical-unit,buffer,word-count,altrtn)

physical-unit	Card punch sub-unit number: 0 CR0, first controller 1 CR1, second controller
buffer	Data name containing line to be punched.
word-count	Number of words to be punched.
altrtn	Alternate return in case of error — never taken in Rev. 19. (See Chapter 14.)

Load Information

This subroutine calls T$PMPC.

▶ O$AC15

Purpose

Punches output to the parallel interface card punch and prints on card.

Usage

CALL O$AC15(physical-unit, buffer, word-count, altrtn)

physical-unit	Card punch sub-unit number:	
	0	CR0, first controller
	1	CR1, second controller
buffer	Data name containing line to be punched.	
word-count	Number of words to be punched.	
altrtn	Alternate return in case of error. (See Chapter 14.)	

Load Information

This subroutine calls T$PMPC.

▶ T$PMPC

Purpose

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 also be called by the user.

Usage

CALL T$PMPC (physical-unit, LOC(buffer), word count, inst, status)

physical-unit	Card punch sub-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 (INTEGER*2). 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, inst would be an octal 12400 meaning:

1. Feed a card.
2. Punch a card.
3. Stack a card.

status Three word status vector:

status(1) Not used.

status(2) Device status returned for a read request (instr = :4000):

Value	Condition
:200	Online
:4	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. Their functions are shown in Table 19-3.

Note

Most of the following subroutines are obsolete and have been replaced with T$MT.

Table 19-3
Functions of Magnetic Tape Subroutines

	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

PRIMOS supports record sizes up to 6K words for 9- and 7-track tapes. Under PRIMOS II, larger records may be used only if the program declares its own labeled common area 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. (See Chapter 15.)

Since the subroutines are similar, they are described in groups.

▶ C$M05, C$M10, C$M11, C$M13

Purpose

These subroutines provide control functions for tape as shown in Table 19-3.

Usage

```
      (C$M05)
      (C$M10)
CALL  (C$M11)  (key, name, physical-unit, altrtn)
      (C$M13)
```

key User option:

-4	Rewind to BOT (Beginning of Tape).
-3	Backspace one file mark.
-2	Backspace one record.
-1	Write file mark.
1	Open to read.
2	Open to write.
3	Open to read/write.
4	Close. (Write file mark and rewind).
5	Move forward one record.
6	Move forward one file mark.

7 Rewind to BOF (Beginning of file).

8 Select device and read status.

name Not used (may be anything).

physical-unit 0-7 (0-3 for PRIMOS II), depending on which device is ASSIGNed).

altrtn The alternate return. (See Chapter 14.)

Discussion

These routines call T$MT and ERRSET.

Error Messages

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

▶ O$AMxx, I$AMxx, O$BMxx, I$BMxx

Purpose

These subroutines provide read and write functions for magnetic tape as shown in Table 19-3.

Usage

These subroutines all have the same calling sequence:

CALL subroutine (physical-unit, buffer, n, altrtn)

physical-unit Sub-unit number = 0, 1, 2, or 3.

buffer Data name from or to which information is tranferred.

n Number or words to be read or written. If _n_ = 0, then the subroutine is to write a file mark.

altrtn FORTRAN alternate return. (See Chapter 14.)

Error Messages

(See Appendix E for ERRVEC.)

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.

Discussion

These subroutines all call T$MT and ERRSET.

▶ T$MT

Purpose

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 to the calling program. If T$MT is called with the code argument, the appropriate error code will be returned to the calling program.

Usage

CALL T$MT (unit, buff, nw, instr, statv [, code])

unit — Magnetic tape drive — logical drive number 0 through 7 (INTEGER*2).

buff — Location of the buffer from which to read or write a record of information (INTEGER*4). It must be an octal number. If neither a read or write operation, buff is 0.

nw — Number of words to transfer. This number must be between 0 and 6K words (INTEGER*2). 6K words can be transferred under PRIMOS only if the buffer starts on a page boundary. Otherwise, the maximum size is reduced by the offset of the buffer from the page boundary.

instr — The instruction request to the magnetic tape drivers (INTEGER*2). Valid instructions are:

Octal	Hexadecimal	Meaning
000040	0020	Rewind to BOT, 7- or 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- or 9-track, and get status.
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.

140000	C000	Return controller id. (See the section on controller id below.) 18.1

Note

The following instructions are only valid with version 2 or 3 (in some cases both versions) magnetic tape controllers. In error situations, if no code argument is given, use of these instructions with older versions of the controller will cause an error message to be printed and the program will be aborted. A description of use of these commands is found later in this chapter.

Octal	Hexadecimal	Meaning
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).
100060	8030	Set density to 800 bpi (version 2 controller only).
100100	8040	Set density to 1600 bpi (version 2 and 3 controller).
100120	8050	Set density to 6250 bpi (version 3 controller).
100140	8060	Enable front panel density select switch (version 3 controller). Set density to 3200 bpi (for future use).
100160	8070	Set speed to 25 IPS (for future use). 19
100200	8080	Set speed to 100IPS (for future use).
043500	4740	Read record backwards (version 3 controller)

18.1 statv — 6-word status vector. If this is the last argument, then only the first three words are set. If the code argument follows, then additional words may be set, depending on the controller being used. The words are:

statv(1) Status flag:

Bits	Meaning
1	Operation in progress
0	Operation finished

statv(2) Hardware status word from controller. Possible values are:

Bits	Meaning
01	Vertical parity (read) error
02	Runaway
03	CRC error
04	LRC error
05	False gap or insufficient DMA range
06	Uncorrectable error
07	Read and correct operation failed
08	File mark detected
09	Transport ready
10	Transport online
11	End of tape detected
12	Selected transport rewinding
13	Selected transport is at load point (beginning of tape)
14	Tape write-protected (file- protected)

15 DMX overrun or no formatter

16 Rewind complete (This bit has no function with version 2 controller.)

statv(3) Number of words transferred (read and write operations only).

statv(4) Hardware status for version 1, 2, and 3 controllers. Bits 0 and 1 specify density of tape:

00	800 bpi
10	1600 bpi
11	6250 bpi

18.1

statv(5-6) Reserved.

code

Specifies that the appropriate error code is to be returned to the calling program. If this argument is omitted, then any illegal instructions will result in an error message being printed, followed by a return to command level (PRIMOS). If this argument is used, then statv must be a six-word array.

The possible error codes returned are:

E$NASS Device specified in physical-unit, not assigned.

E$IVCM Invalid command (e.g. attempt to set density on version 0 controller).

E$DNCT Device specified in physical-unit not connected, or no controller.

E$BNWD Invalid number of words (nw <=0 or >6144).

Discussion

Magnetic tape I/O is not buffered under PRIMOS. A call to T$MT returns immediately before the operation is complete. When the magnetic tape operation is completed, the status flag in the user space is set to 0. Therefore, a user program may do another computation while waiting. If a user initiates another call to T$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. The drive should be set to the right density first.

For example, if the user set the density to 6250 bpi with the ASSIGN command and read the first record of a 1600 bpi tape, then the rest of the tape would be read using 1600 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 1600 bpi. If the tape was rewound and then a record was written, the density would be switched to 6250 bpi. Although the density setting of 6250 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 1600 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.

Instruction to Get Controller Id

18.1 The controller id may be used by software that intends to support all tape drives, but takes advantage of special features that are available only with a particular controller. For example, the ERASE command is only available with version 2 and 3 controllers.

Figure 19-1 shows how buf(1) must be set up for this instruction (:140000).

```
|0              8|9            16|
----------------------------------
|  not used     |  Contr. ID*   |
----------------------------------

* ID from Table 19-4
```

BUFF(2) When instr is :140000
Figure 19-1 18.1

Table 19-4
Controller Id

Version	Device ID	Controller #	Drive Type
0	'014	2081	Pertec
1	'114	2081	Kennedy, separate formatter
2	'214	2269/2270	Kennedy, two-board integrated controller
3	'314	2023	Telex(1600/6250 bpi)

Use of the T$MT Wait Semaphore

While waiting for an operation to complete (that is, for status-word 1 to go to 0), a process can do one of several things. It can loop while checking the status-done word, do another operation (such as get status), or use a 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. See Chapter 21 for a discussion of semaphores.

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 CODE, CODE2       /* RETURN CODES
       INTEGER STATV(6)          /* STATUS VECTOR SET BY T$MT
       INTEGER UNIT              /* MAG TAPE DRIVE NUMBER (0-7)
       INTEGER BUF (1024)        /* OUTPUT BUFFER
       INTEGER XSTATV (6)        /* SCRATCH VECTOR FOR WAIT

       . . .
       CALL T$MT (UNIT, LOC(BUF), ,:042620,STATV, CODE)
                                 /*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, CODE2) /* 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 3 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. 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.

3. Attempt to rewrite the record.

4. If the record was not written successfully, repeat steps 2 and 3 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 VI

Communications Controllers and Realtime Subroutines

20 Synchronous and Asynchronous Controllers

This chapter presents the following subroutines:

Routine	Function
T$SLCO	Communicate with SMLC driver.
ASNLN$	Assign AMLC line.
T$AMLC	Communicate with AMLC driver.

19

SYNCHRONOUS CONTROLLERS

This section defines the raw data mover for the assigned SMLC line. See the System Administrator's Guide for a discussion of SMLC lines.

▶ T$SLCO

Purpose

The SMLC driver is loaded in PRIMOS. A user program communicates with the driver via FORTRAN-format calls to T$SLCO. The driver communicates with the user address space via buffers in the user address space specified by the user program. The data structure used by the driver is a control block created by the user in the user address space. It contains pointers to the user status buffer and to buffers containing a message to be transmitted or set to receive a message. A separate control block is required for each line.

Usage

```
CALL T$SLCO (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 Tables 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 one 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 equal 1. The possible model numbers and their associated protocols are the following.

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

line Octal line number 0-7. | 18.1

LOC(block) Address of user's block. User's block must reside entirely within one page.

nwds Number of words in block.

Discussion

Before calling T$SLCO to configure a line (key = 3), a call with (key = 7) should be made to see if the Multiline 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.

Timing

The user space program runs asynchronously with message transfers. A call to T$SLCO 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:

$$\text{ASSIGN} \left\{ \begin{matrix} \text{SMLC00} \\ \text{SMLC01} \\ \text{SMLC02} \\ \text{SMLC03} \\ \text{SMLC04} \\ \text{SMLC05} \\ \text{SMLC06} \\ \text{SMLC07} \end{matrix} \right\}$$

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 (continued)
Key = 2 SMLC Control Block

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:	1 Transmit 0 Receive

Note

If Word 8 is given (nwds > 8) then at least one DMC address pair must be given.

Words 9-10 11-12 13-14 15-16	DMC start and end address pointers. Up to four pairs may be specified to allow for channel chaining.

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 (BISYNC, UT200, ICL7020, 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
            |     |     |
            |     |     └─Automatic parity-enable
            |     └───────Parity-select 0 = odd,*
            └─────────────Parity-enable

Bits 15-16  15    16
             \___ ___/
                 └──────Number of bits per character
```

If automatic parity is enabled with 8-bit data enabled, no parity will be generated or checked (i.e., no 9-bit data formats).

*Automatic parity-enable appends a parity bit to the data while parity-enable steals the most significant bit of each data byte.

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
                       \___ ___/  |  |
                           |      |  |_Select formatter mode:
                           |      |         0  EBCDIC
                           |      |         1  ASCII
                           |      |
                           |   Select BCC:
                           |      1  LRC (for use with ASCII mode only)
                           |      0  CRC-16
                           |
                   Unused control bits
```

Table 20-4
Key = 3 Line Configuration Control Block (5646, Bits 1-9)

5646
BISYNC

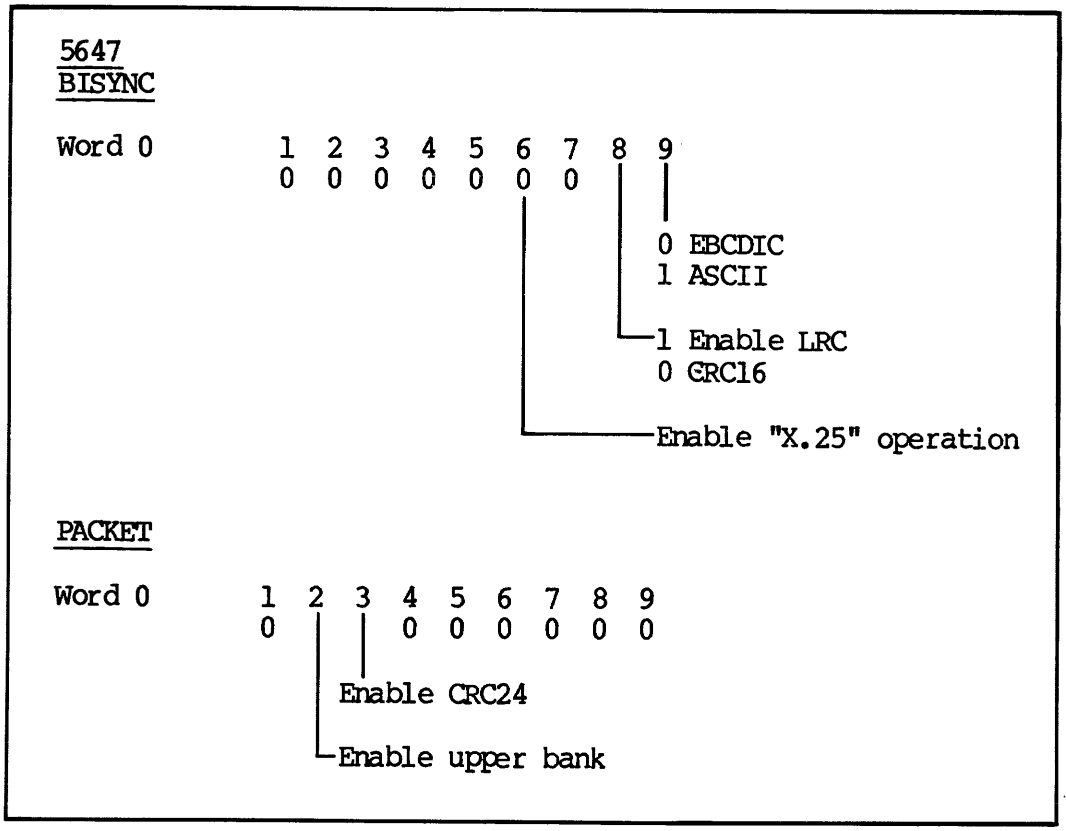
Word 0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	Enable "X.25" operation	0	1 Enable LRC 0 CRC16	0 EBCDIC 1 ASCII

HDLC

Word 0	1	2	3	4	5	6	7	8	9
	Enable secondary station mode.	1	0	Enable all-parties address mode.	HDLC enable.	Tx: Start on right byte. Rx: Start on right byte and generate encoded status if message ends with the left byte.	Enable GO-AHEADs (loop mode).	Tx: 0 = FLAG line during idle periods. -1 = MARK line during idle periods.	Tx: End message on left byte.

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        1  2  3  4  5  6  7  8  9
              0  0  0  0  0  0  0  |  |
                                |  |  |
                                |  |  0 EBCDIC
                                |  |  1 ASCII
                                |  |
                                |  └──1 Enable LRC
                                |     0 CRC16
                                |
                                └─────Enable "X.25" operation

PACKET

Word 0        1  2  3  4  5  6  7  8  9
              0  |  |  0  0  0  0  0  0
                 |  |
                 |  Enable CRC24
                 |
                 └─Enable upper bank
```

Table 20-6
Key = 3 Line Configuration Control Block (5650, Bits 1-9)

5650
BISYNC

```
Word 0     1 2 3 4 5 6 7 8 9
           0 0 0 0 0 | 0 | |
                     |   | └─0 EBCDIC
                     |   |   1 ASCII
                     |   |
                     |   └───1 Enable LRC
                     |       0 CRC16
                     |
                     └───────Enable "X.25" operation
```

ICL7020/UT200/1004

```
Word 0     1 2 3 4 5 6 7 8 9
           | 1 0 0 0 | 0 1 1
           |         └─Enable ICL7020*
           |
        Enable 1004*
```

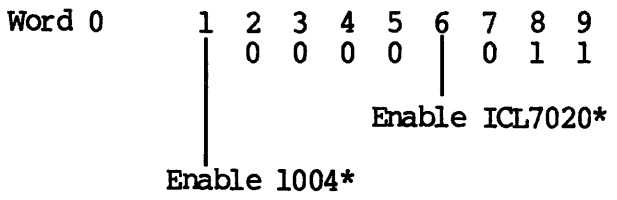
Recommended Configurations

1004	'140722	
UT200	'40723	(Add '40 to enable DSS interrupts.)
ICL7020	'42723	

* Default protocol is UT200

Table 20-7
Key = 3 Line Configuration Control Block (5651, Bits 1-9)

5651
ICL7020/UT200/1004



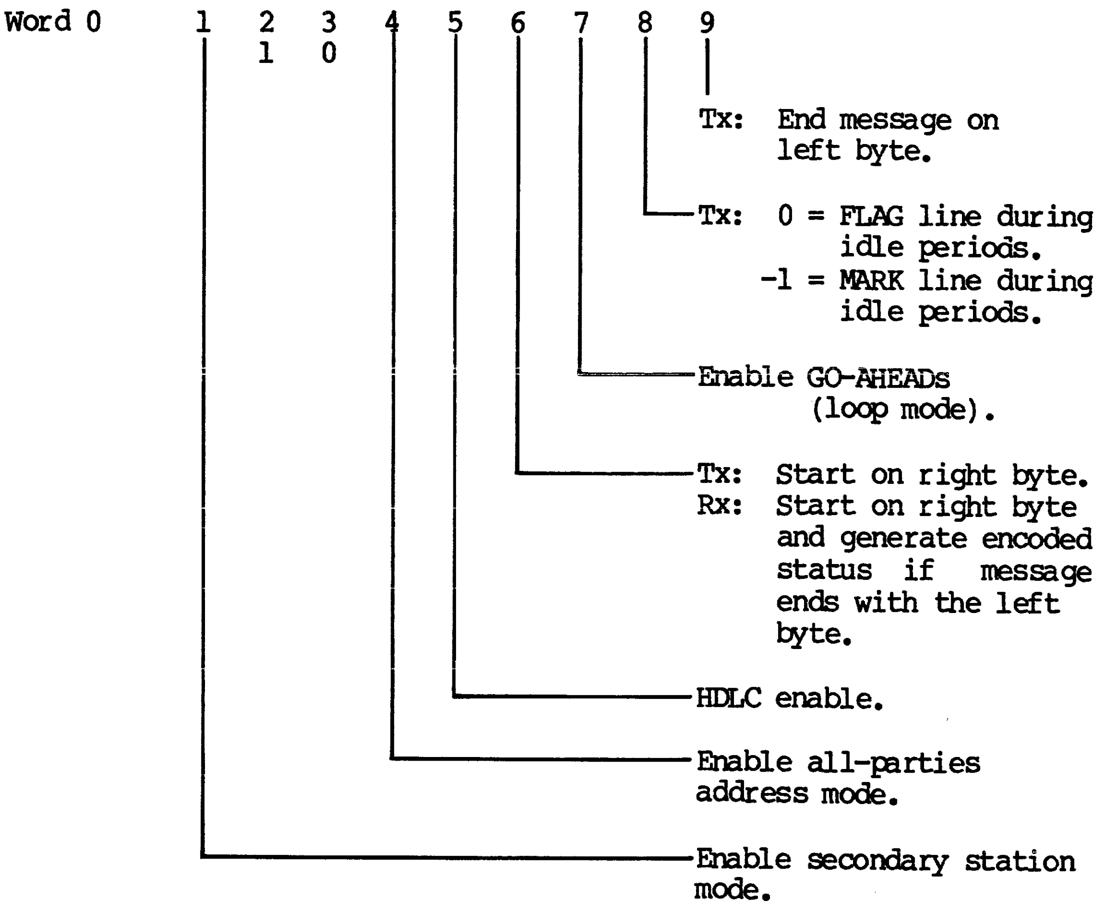
Enable ICL7020*

Enable 1004*

Recommended Configurations

UNIVAC	'100722	
UT200	'723	(Add '40 to enable DSS interrupts.)
ICL7020	'2723	

HDLC



Tx: End message on left byte.

Tx: 0 = FLAG line during idle periods.
-1 = MARK line during idle periods.

Enable GO-AHEADs (loop mode).

Tx: Start on right byte.
Rx: Start on right byte and generate encoded status if message ends with the left byte.

HDLC enable.

Enable all-parties address mode.

Enable secondary station mode.

Secondary station mode, HDLC mode, loop mode, and all-parties address mode are enabled on a line-pair basis only.

*Default protocol is UT200

Table 20-8
Key = 3 Line Configuration Control Block (5652, Bits 1-9)

```
5652
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 (UT200=Default)

                  Recommended Configurations

                  1004       '100722
                  UT200         '723       (Add '40 to enable
                  ICL7020      '2723            DSS interrupts.)


PACKET

Word 0      1   2   3   4   5   6   7   8   9
            0   |   |   0   0   0   0   0   0
                |   |
                |   Enable CRC24
                |
                └──Enable upper bank
```

Table 20-9
Key = 3 Line Configuration Control Block (5653, Bits 1-9)

5653
HDLC

Word 0	1	Enable secondary station mode.
	2	0
	3	0
	4	Enable all-parties address mode.
	5	HDLC enable.
	6	Tx: Start on right byte. Rx: Start on right byte and generate encoded status if message ends with the left byte.
	7	Enable GO-AHEADs (loop mode).
	8	Tx: 0 = FLAG line during idle periods. -1 = MARK line during idle periods.
	9	Tx: End message on left byte.

Secondary station mode, HDLC mode, loop mode, and all-parties address mode are enabled on a line-pair basis only.

PACKET

Word 0	1	0
	2	1
	3	Enable CRC24
	4	0
	5	0
	6	0
	7	0
	8	0
	9	0

Table 20-10
Key = 3 Line Configuration Control Block (5654, Bits 1-9)

```
5654
BISYNC

Word 0     1   2   3   4   5   6   7   8   9
           0   0   0   0   0   |   0   |   |
                                   |   |   |
                                   |   |   0  EBCDIC
                                   |   |   1  ASCII
                                   |   |
                                   |   1 Enable LRC
                                   |   0 Enable CRC16
                                   |
                                   └─────Enable "X.25" operation


GRTS

Word 0     1   2   3   4   5   6   7   8   9
           0   1   0   0   0   |   0   |   |
                                   |   |   |
                                   |   |   0 EBCDIC
                                   |   |   1 ASCII
                                   |   |   GRTS uses ASCII
                                   |   |
                                   |   1 Enable LRC
                                   |   0 Enable CRC16
                                   └──────GRTS uses LRC

                                  Enable "X.25" operation
                                  not used in 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 transmitter
		0	Disable transmitter
	Bit 16:		
		1	Enable receiver
		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 movers for assigned AMLC lines. Refer to the System Administrator's Guide for the AMLC command and how to assign AMLC lines.

▶ ASNLN$ (Assign AMLC line)

Purpose

ASNLN$ allows user programs to request the assignment of a line directly.

Usage:

```
DCL ASNLN$ (FIXED BIN, FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN,
            FIXED BIN);

CALL ASNLN$ (key, line, protocol, config, lword, status)
```

status — Error status returned to caller.

19 key — Assignment option:

- 0 Unassign AMLC line.
- 1 Assign AMLC line.
- 2 Unassign all AMLC lines owned by caller.

line — Desired line number.

protocol — Desired protocol (input and output). Blanks indicate no change desired. The default is TRAN (transparent).

config — Desired config setting. 0 indicates no change desired.

lword — Desired line characteristics. The buffer number used for the line cannot be changed by a user program using this interface.

Description

This routine is a new direct entrance call available to users. It performs the assignment and unassignment of AMLC lines for a caller. A user may own more than one assigned line. The caller may also set line characteristics, protocol, etc. This routine will only allow a caller to assign a line that has a corresponding LBT entry of 0, which means that the line is assignable. The buffer used for the assigned line is dynamically chosen within ASNLN$. 19

Refer to the System Administrator's Guide for protocol, config, and lword values.

▶ T$AMLC

Purpose

T$AMLC is a direct entrance call. It performs raw data movement, provides status information about assigned AMLC lines, and transfers characters between the caller's buffer and a desired assigned line's buffer. The caller must own the desired line, that is, the corresponding LBT entry must contain the caller's user number.

Usage

```
DCL T$AMLC (FIXED BIN, PTR, FIXED BIN, FIXED BIN, FIXED BIN,
            FIXED BIN, FIXED BIN);

CALL T$AMLC   (line, user-buf-addr, char-count, key, stat-vec,
               char-pos-arg, errcode)
```
19

line — Desired AMLC line number.

user-buf-addr — Address (pointer) to the caller's buffer.

char-count — Desired number of characters to move. No maximum limit is enforced.

key — Desired function:

1 Input char-count characters.

2 Input char-count characters or until .NL. is encountered. stat-vec(1) will be the actual number of characters read.

3 Output char-count characters. Maximum is char-count. This key assures the caller that char-count characters will be output. For example, an error is not returned if the line's input or output buffer is smaller than char-count. T$AMLC will output blocks of data from the caller's buffer into the available room in the line's output buffer until char-count is exhausted. A one-second wait is issued between output chunks to allow time for the line's output buffer to clear. In most cases, the entire char-count should be output at once.

4 stat-vec(1) = number of characters in input buffer. stat-vec(2) = state of carrier. 0 = carrier, not 0 = no carrier.

5 Return status of output buffer. stat-vec(1) = 1 if room for char-count in output buffer. stat-vec(1) = 0 if not enough room for char-count. stat-vec(2) = state of carrier. 19

6 Input all available characters in the input buffer. Maximum = char-count. This key will place all the available characters in the line's input buffer into the caller's buffer. stat-vec(1) = number of characters actually input.

7 Return additional output buffer status. (Refer to key 5.) stat-vec(1) = amount of character space remaining in the output buffer.

8 Flush input buffer.

9 Flush output buffer.

10 Flush both output and input buffers.

11 Output characters to available room in output. This key will output as many characters as possible into the line's output buffer. A wait will not be done to exhaust char-count. stat-vec(1) = char-count minus the number of characters actually output.

stat-vec(1) = number of chars that were not successfully output. If stat-vec(1) = 0, this means all characters were output.

stat-vec Two-word status vector used by certain keys.

char-pos-arg The caller may wish to indicate a starting position within the buffer addressed by user-buf-addr. Char-pos-arg applies for both input and output keys. 19 This is an optional argument. If omitted, the default is to start with the first character. Note: if char-pos-arg is used, the first character position should be indicated by 1 (there is no character at position 0). Also, char-pos-arg is not updated within T$AMLC.

errcode Optional argument to return error status. If errcode is present, error messages will not be printed at the caller's terminal.

21 Semaphores and Timers

REALTIME AND INTERUSER COMMUNICATION FACILITIES

PRIMOS supports user applications that have realtime 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 realtime applications and interuser synchronization is in the form of a set of subroutines that provide access to Prime's semaphore primitives (wait and notify) and to internal timing facilities.

Table 21-1 lists the subroutines available for handling these facilities.

SEMAPHORES

On timesharing systems where more than one process can be active at the same time, there is often a need to coordinate the execution of multiple processes with one another. Such coordination is required when two or more processes cooperate to solve a common problem, or when multiple processes must use a common, limited resource.

Table 21-1
Semaphore Subroutines by Function

Open (Request) Semaphore
SEM$OP (by filename) (2)
SEM$OU (by file unit) (2)

Notify Semaphore
SEM$NF

Wait
SEM$WT

Test Counter
SEM$TS

Drain (Reset Counter or Notify)
SEM$DR

Set Timer
SEM$TN (1)

Timed Wait
SEM$TW (2)

Close Semaphore
SEM$CL (2)

Suspend Process
SLEEP$

Notes to Table 21-1

1. For numbered semaphores only
2. For named semaphores only

When multiple processes are working together as part of a larger system or to solve a common problem, it sometimes happens that one or more of the processes encounter a situation in which they cannot do any further work until some event, external to the process, happens. An example of this is a spooler which picks up print requests from a queue. When there are requests in the queue, the spooler services them; however, when the queue becomes empty, it can no longer do useful work and must wait for another process to give it something to do.

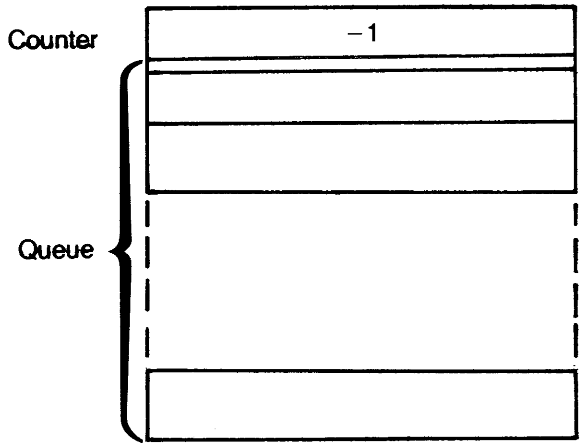
There are many resources on a timesharing system that must be shared by all of the running processes. Included in the list are such things as devices that can have only one user at a time (such as a paper-tape punch), a section of code that performs a single operation, or files that are updated and read simultaneously by several programs.

The semaphore facility provides a means to coordinate multiple processes, providing that the processes involved all use the facility in the same way.

The semaphore facility consists of some blocks of memory, which are called semaphores, and a set of software routines or hardware instructions that perform various operations on these blocks. There is no real connection between a semaphore and the event or resource with which it is associated. The use to which a semaphore is put is determined solely by the application programs that use it. All of the cooperating programs must agree on the meaning (or use) of a semaphore and use it the same way.

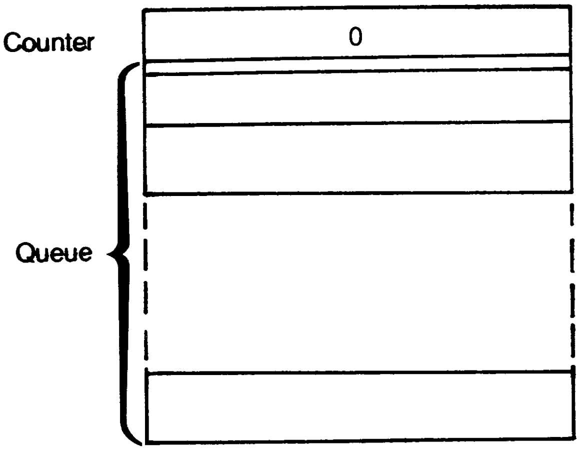
How a Semaphore Works

A semaphore consists of two parts: a counter and a queue.



Resource Semaphore at Start
Figure 21-1

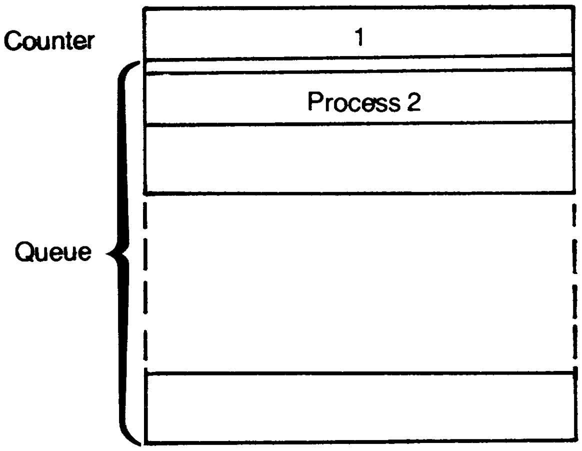
When a process wishes to wait for an event to happen or a resource to become available, it issues a wait call for the semaphore associated with that event or resource. The wait call will increment the counter for that semaphore and test its value. If the counter is less than or equal to 0, the process is allowed to proceed immediately and is not placed on the semaphore's queue.



Resource Semaphore After Call by One Process
(Process 1 is Using the Resource, No Processes Waiting)

Figure 21-2

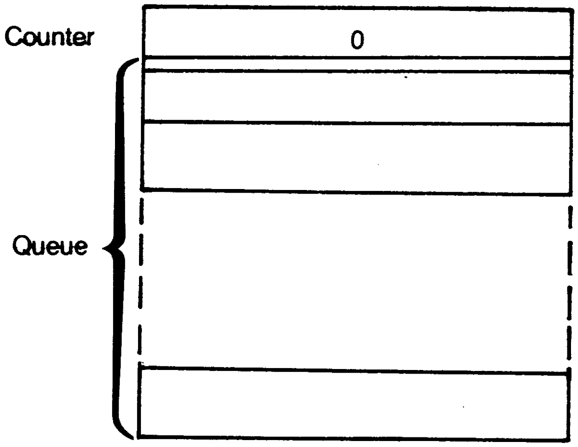
If, however, the counter is greater than or equal to 1 after being incremented, then the process is placed on the wait queue for the semaphore. The process will not run again until it leaves this queue. Processes are placed on the queue in priority order with higher priority processes being placed closer to the head of the queue. Within a given priority, the processes are treated as a real queue — first in, first out.



Resource Semaphore After Call by Second Process
(First Process is Using the Resource)

Figure 21-3

When a process wishes to report that an awaited event has occurred, or that a resource has become available for use by other processes, it will call a notify routine for the semaphore associated with that event or resource. The notify routine will first test the value of the counter for that semaphore. If the counter is greater than 0 (indicating that one or more processes are in the semaphore's queue), then the routine will remove one process from the top of the queue, thereby allowing that process to run again. Whether a process was dequeued or not, the routine will then decrement the counter by one.



Resource Semaphore After Notify by One Process
(Process 2 is Now Using the Resource)

Figure 21-4

Normally, a semaphore's counter is preset to some value before the semaphore is used by any process. The value to which it is set depends on the nature of the software that will use the semaphore and on the purpose of the semaphore. Typical initial values are -1 and 0. A value of -1 allows the first process that waits on the semaphore to proceed immediately without being queued, as shown in Figures 21-1 through 21-4. This effect is desirable if the semaphore is used to coordinate the use of a shared resource. The resource is considered available until a process indicates its intent to use it. A value of 0 is appropriate for wait situations in which a process must wait until some condition exists or until an event occurs. The process that must wait for an event to happen does a wait operation on the semaphore, and is immediately put on the queue since the counter becomes greater than 0. When another process determines that the awaited event has occurred, it will notify the same semaphore, thus allowing the queued process to run.

When a process opens a named semaphore, and that process is the first to open that semaphore, then the SEM$OP routine will preset the semaphore's counter to a value of 0. If an initial value of -1 is required, then the process should notify the semaphore once after opening it. For named semaphores, SEM$OU also allows opening semaphores with initial values that are negative or 0. The minimum value is -32767. If the semaphore must be reset to its initial value of 0 at a later time, then a call can be made to the drain routine (see SEM$DR below).

Cooperation of Processes

It should be remembered that a semaphore is a structure that cooperating processes can use to control their access to resources, or to coordinate their execution. The operating system does not verify that the semaphore is being used correctly since the association between the semaphore and the event or resource is merely a convention adopted by the processes involved.

In order for the semaphore facility to work correctly, all processes that want to wait for an event or a resource must first wait on its associated semaphore before using the resource or assuming that the awaited event has occurred. There is nothing to stop the careless programmer from using a shared resource without first waiting on the appropriate semaphore. Such coding practices will most likely cause the entire subsystem of processes to malfunction.

PRIME SEMAPHORES

On Prime computers, a semaphore consists of two (16-bit) consecutive, nonpageable words of memory. The wait and notify operations are implemented in firmware and are usable by supervisor software only. So that users can use the semaphore facility, four calls have been created that perform the wait and notify operation on a set of semaphores that are reserved by the operating system for user programs:

- SEM$WT
- SEM$TW
- SEM$TN
- SEM$NF

In Rev. 19 there are 1024 named semaphores available to user processes, and 65 numbered semaphores.

Numbered Semaphores and Timers

Internal to PRIMOS is an array of 65 numbered semaphores reserved for the use of user processes. All reference to these semaphores is by the index of the semaphore, an integer from 1 to 65. 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 65 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.

Numbered semaphores are assigned by the operating system as wait or notify calls are made involving those numbers. No open or close request is necessary. It is the programmer's responsibility to use the number that has been agreed upon for a particular resource.

Named Semaphores

The operating system maintains a pool of semaphores which it can assign to user processes. When a process wishes to use one or more named semaphores, it must first ask the operating system to assign them to the process. The process requests access to named semaphores via an open routine. The user can request that multiple semaphores be assigned to it in a single call to this routine. The operating system will return a set of numbers to the process if it decides that the requested semaphores can be assigned to that process. The process will use these numbers in all subsequent calls to semaphore routines to indicate on which semaphore to perform the semaphore operation.

The operating system can tell when different processes wish to use the same set of semaphores by examining the parameters that they include in the call to the open routine.

(See SEM$OP and SEM$OU below for more details on how to use the open call.)

After a process has opened a set of semaphores, it can do any number of operations on those semaphores. The possible semaphore operations are described in the section entitled DESCRIPTION OF THE SUBROUTINES.

When a process has finished using the named semaphores that were assigned to it, it requests that the operating system close those semaphores, thus making them inaccessible to the process. When all processes that were using a semaphore close it, then the space in the operating system taken up by that semaphore is returned to the operating system's free pool and may be assigned to other processes at a later time.

When a process logs out, all named semaphores that were opened by the process but not closed are closed automatically. If this process was the last user of a semaphore, the space used by the semaphore is returned to the free pool.

CODING CONSIDERATIONS

Named vs. Numbered Semaphores

There are two methods by which a process can specify which semaphores it intends to use. Also, there are two sets of semaphores maintained by the operating system. One set is available to any process that wishes to use it, and its semaphores are identified by number. When a process wishes to use one of these semaphores, it specifies the number of the desired semaphore in the parameter list of the semaphore routines. This set of semaphores is called numbered semaphores. Numbered semaphores are easy to use, but they have a major drawback: there is nothing to prevent other processes from using the same semaphore for different purposes. Therefore, all users of the system must agree on the usage that each numbered semaphore will have; otherwise, confusion will result.

To eliminate the problems caused by the sharing of numbered semaphores, a second set of user semaphores was created. These are called named semaphores because they are associated with a file. Semaphores in this set cannot be used by a process until they are opened. Opening a semaphore means that the process must call the routine SEM$OP or SEM$OU, which will assign semaphores from the pool for the process to use. Each routine returns a set of numbers which can be used instead of numbered semaphore numbers in all other semaphore routine calls. Only valid semaphore numbers that have been assigned to a process by SEM$OP or SEM$OU can be used in subroutine calls that manipulate named semaphores. An attempt to use any other numbers will result in an error return from the routine.

To open a set of named semaphores, a routine must associate them with a file system object. SEM$OP will open a set of named semaphores and associate them with the name of a file in the current UFD of the process performing the open operation. SEM$OU will open a set of named semaphores and associate them with a file open on a particular file unit. In both cases, the process must have read access to the file.

Timers and Timeouts

When a process waits on a semaphore, it anticipates that it will be notified within a reasonable amount of time. If, for some reason, the process that is going to notify the semaphore fails to do so, all processes waiting on that semaphore will continue to wait, possibly for a very long time.

To guard against processes waiting forever, a timer mechanism is used.

Named Semaphore Timers: To prevent a process from waiting forever on a named semaphore, a special wait routine exists (called SEM$TW) which takes a semaphore number and a time value as parameters. The process will wait on the specified semaphore until the semaphore is notified or until the specified amount of realtime has passed. The routine returns a value to the process that indicates why the process was allowed to continue. A value of 0 means that the semaphore was removed from the wait queue because of a notify by another process. A value of 1 means that the process was allowed to continue because the specifed time had elapsed without a notify on that semaphore. It is also possible for a value of 2 to be returned; this return value indicates that the process was stopped by someone pressing the BREAK key or CONTROL-P at the terminal controlling the process, and then typing START. This sequence causes the operating system to abort the process, thus removing it from the semaphore on which it was waiting, followed by a restart of the process at the wait call.

Numbered Semaphore Timers: The timer facility for numbered semaphores allows a semaphore to be automatically notified after a certain amount of time has passed. A user process tells the operating system, via a subroutine call, that a timer is to be associated with a numbered semaphore. The process also specifies the amount of time that should pass before the operating system notifies the semaphore. When this amount of time has passed, the operating system notifies the semaphore.

Much care is needed when coding programs that use semaphores with this kind of timer. If another method is not used besides the semaphore to indicate that the awaited event has actually occurred, then a notify caused by a timer cannot be distinguished from a notify caused by a process. The processes using the semaphore should, therefore, be coded so that they can verify that a notify by another process has occurred before using the resource protected by the semaphore. The action that is taken when a timer notifies the semaphore should be agreed upon by all of the processes using the timed semaphore.

PITFALLS AND HOW TO AVOID THEM

External Notifies

When a semaphore is notified for some reason other than an explicit call to the notify routine, that notify is called external; that is, it originated from a source external to the processes that are using the semaphore. Some of the reasons that an external notify may occur are listed here.

Expiration of a Timer: When a timer is set for a numbered semaphore, and that timer expires, the operating system will notify the semaphore.

This semaphore will look like an external notify to the processes that use the semaphore; the fact that the notify is external can be detected if the processes are coded properly. (See Coding Suggestion below.)

The notify caused by a timeout can be useful in cases when the process that is supposed to notify the semaphore is prone to being aborted. The notify initiated by the operating system will prevent processes from waiting forever.

Use of timers with named semaphores causes a code to be returned to the process that indicates when a timeout has occurred.

Malfunctioning Process: Processes that are supposed to be using a semaphore, like all other programs, sometimes do not behave properly. Malfunctioning programs can do extra notify calls, and cause what appear to be external notifies. Also, processes that are not supposed to be using a numbered semaphore may decide to use it anyway. Unless the semaphore can be protected from such interference, then what appears to be an external notify will result.

Process Quit: The semaphores that a user process can access on a Prime system are called quittable semaphores. This means that a process that is waiting on a semaphore can be stopped by pressing the BREAK key or CONTROL-P at the terminal controlling the process. When a process is stopped by this means, and then continued (by using the PRIMOS START command), the process will reexecute the wait operation.

Coding Suggestion: Since semaphores can be notifed by breaks and timeouts as well as by explicit calls to SEM$NF, and since this could cause malfunctions in a subsystem, it is always best to code in such a way that this situation can be detected. This means that a process should not rely solely on the semaphore to indicate that a resource is really available or that an event has actually occurred. A good practice is to have one additional method, besides the semaphore, to indicate what the current state of the resource or event is.

One such method is to have a word in shared memory (accessible by all cooperating processes) which is set to indicate that the event has really occurred or that a resource is free. Before a process notifies a semaphore, it sets this word to an agreed value. When the process is allowed to proceed from a semaphore wait, it should check the value contained in that word. If the word contains the value, it will know that the semaphore was notified by a cooperating process, and not by the operating system. In this case, the process will clear the word, do its processing, and reset the word to the agreed upon value just before notifying the semaphore. If a process proceeds from a wait call and the word is not set to the agreed upon value, it can assume that the operating system notified the semaphore and can reissue the wait call.

Infinite Waits

It is possible to create a situation in which one or more processes are waiting on a semaphore, and there are no processes running that will ever notify that semaphore. The following are methods of creating this situation.

Multiple Waits: If a process issues a wait call, and is not queued, and then continues to reissue the wait call without intervening notifies, that process will eventually cause the semaphore count to become greater than 0 and the process will wait. This of course assumes that there is not another process somewhere doing multiple notifies.

In the case of a resource-protection semaphore, if all other processes obey the rules, they will wait on this semaphore before they notify it. They will therefore queue up behind the multiple-waiter process. Eventually, all the processes of the subsystem will become queued on the semaphore queue, and no process will remain to notify the semaphore.

Aborted Notifiers: Another way of causing infinite waits is to abort a process that would, under normal circumstances, notify a semaphore. If this is the only process that will do notifies on the semaphore, then all other processes that wait on it will wait forever.

Coding Suggestion: Infinite waits can be avoided by associating a timer with the semaphore. This will guarantee that one or more processes will eventually be removed from the wait queue. Extra coding must be done in the processes, however, so that a timeout can be identified as such, and so that appropriate action can be taken. This code should determine whether the process that should have notified the semaphore is still running or not. If it is running, the notify is considered external and the process reissues the wait call. If the potential notifiers have all been aborted, appropriate recovery action should be initiated.

Deadly Embrace

When multiple semaphores are being used, a situation called deadly embrace can occur. Ths happens when two processes each gain rights to use a resource by waiting on the appropriate semaphore for that resource, and then each attempts to acquire the resource that is being used by the other process. Clearly, neither process will ever notify the semaphore for the resource it holds (it is waiting to get access to a second resource), and no other process will ever notify the semaphores (since each resource is held already by one of the two processes). Therefore, both processes will wait forever.

This situation can neither be detected nor prevented by the semaphore facility. It can be prevented, however, by the processes using the semaphores, if the following procedure is used.

Each semaphore that a system of processes will use is assigned a different number; this number will be called the semaphore's level number. Processes can only issue a wait call for a semaphore whose level number is greater than the level number of any semaphore it has waited on but has not yet notified. For example, if the level numbers for three semaphores are 1,2, and 3, and a process has waited on the second semaphore (level 2), but has not yet notified it, then the process can legally issue a wait for the third semaphore (level 3) but not for the first, since level 1 is numerically less than level 2.

This technique, if strictly followed, makes deadly embrace situations impossible. It is sometimes practical for processes to call a routine which checks for level number violations before issuing a wait call. If all processes use this routine instead of the wait routine then deadly embrace is prevented.

LOCKS

Locks, like semaphores, are a method which programs or processes can use to coordinate their usage of some resource. Before a process attempts to use a resource that is protected by a lock, it calls a routine that grants or denies permission to use the resource or causes the process to wait until the resource becomes free. When the process has been given permission to use the resource, it is said to hold the lock on that resource. When the process is through using the resource, it calls another routine to indicate that it is done. This operation is called giving up the lock, or releasing the lock on that resource.

Various types of locks exist, some of which will be discussed in this section.

Some types of locks behave very much like semaphores and, in fact, many types of locks can be coded with the use of semaphores. Semaphores, unlike locks, allow a small, well-defined set of operations to be performed while the uses and types of locks that can be coded vary greatly.

Mutual Exclusion

Mutual-exclusion locks are used when only one or a few processes are allowed to use a resource at any given time. When a process requests ownership of a lock for the resource, it is given the lock if no other process currently holds it. If the lock is held by another process, all others must wait until the one holding the lock gives it up.

This type of lock can be implemented directly with the use of semaphores. Requesting the lock is equivalent to a wait operation on a semaphore; giving up the lock is equivalent to a notify of that semaphore.

Since external notifies may occur, it is a good practice to expect them and to code in such a way that they can be detected and ignored.

N1 Locks

N1 locks are used to protect objects that can be both read and modified simultaneously, such as files and data bases. This type of lock allows any number of users to read the object, or one process to modify the object. When a process requests permission to read the object, such permission is granted immediately, as long as there is not currently a process modifying it. Requests to gain access to the object for modification are granted only if there are no other readers or writers using the object. If another process is using the protected object, the writer is placed on a queue and must wait until all current users of the resource indicate that they are done. If a writer is waiting to use the resource, then no other requests for use of the object are granted until that process has used the object. This prevents readers from gaining access to the object and causing the writer's request to be delayed indefinitely.

When a writer is given access to the object, all other requests for access are queued. When the writer finishes, the other requests are processed.

Use of an N1 lock on a file eliminates data loss that can sometimes occur when multiple processes are allowed to update the same file simultaneously.

Producers and Consumers

In many computer systems, certain processes create work which must be processed, such as device drivers that read data from a device which must be routed to the correct place, or print programs that place data files into spool queues to be printed. These work-producing processes are called producers.

Other processes in a system process the work created by the producers. These processes are called consumers. Examples of consumers include a user process that manipulates data coming into the system from a peripheral device, or a spooler that prints files in response to a user's print requests.

The coordination required between producer processes and their corresponding consumer processes can be achieved with the use of producer-consumer locks.

Producers call a routine that indicates that there is work to process. The routine keeps track of the number of producers that have called it; each call indicates that another unit of work is available. Consumers, on the other hand, call a routine that checks to see if there is any-work-to-do. If there is no work, the routine causes the consumer process to wait until there is work, that is, a producer calls the "I-have-work-to-do" routine. If there is work to do, the consumer process is allowed to continue, and the counter of units of work left to do is decremented.

This lock can be coded directly with semaphores. A semaphore, with its counter initialized to 0, serves as the locking mechanism. Producers notify the semaphore, causing it to become negative; consumers wait on the semaphore, causing it to rise toward 0. If there is no work to do (semaphore counter equal to 0) then a consumer will be queued, when it waits on the semaphore, until work becomes available.

Note that there can be any number of producers or consumers. If multiple consumers wait for work, and there is none to do, then the semaphore counter will contain a value equal to the number of queued consumer processes. A notify by a producer will allow one of the consumers to proceed.

Since semaphores are subject to external notifies, it is advisable that a counter, other than the counter that is a part of the semaphore, be maintained to indicate how much work is available for consumer processes. Producers will increment this counter; consumers will take work from the work queue and decrement this counter. If a consumer is notified out of the semaphore queue and the counter does not match the semaphore counter, then it can assume that an external notify has occurred.

Record Locks

When many processes must update a file, and speed is important, it is not practical to use a lock which protects the entire file, since any update request would lock all other processes out of the file. Considerable overlap in processing can usually be achieved if just the portion of the file that is being updated by a process is locked. Usual units to lock are the record or the page being updated.

If the file is large, then it becomes impractical or impossible to have an individual lock for each record or page to be protected. One way of overcoming this difficulty is to assign locks from a pool on a temporary basis. When a process wishes to update a record, for example, it requests a lock by passing the record number in question to the lock routine. If there is currently no one holding a lock on that record (the lock routine scans its list of locks being held by other processes), then a lock is assigned from a free pool and the record number supplied is remembered. If a lock is requested for a record that is currently locked by another process, then the second and subsequent requesters of the lock are forced to wait. When the last holder of a lock gives up the lock, and there are no other processes waiting to use the record protected by that lock, then the lock itself is returned to the pool of free locks. It can then be used for other record locks.

In general, the pool of locks needs to be as large as the expected maximum number of records that can be locked at any given time. It is the lock routine's responsibility to manage the lock pool and to deal with the problems that arise when there are no more free locks in the pool. One method of dealing with this situation is to use a "no-free-locks" semaphore. If there are no free locks in the pool, the process requesting the lock is forced to wait on this semaphore. The lock routine notifies this semaphore when a lock becomes available.

Notice that record locks are really mutual-exclusion locks; however, the object that is being protected by any given lock changes with time. The lock routine must include a small data base that is used to remember what is being protected by each lock.

DESCRIPTION OF THE SUBROUTINES

The following semaphore operations are available to user processes. Table 21-1 shows the subroutines by function.

▶ SEM$OP

▶ SEM$OU

Purpose

These routines open a semaphore.

Usage

CALL SEM$OP (fname, namlen, snbr, ids, code)

or

CALL SEM$OU (funit, snbr, ids, init-val, code)

funit The number (1-127) of a file unit that has been opened (FIXED BIN).

fname A filename, discussed below (char(32)).

namlen The number of characters in fname (FIXED BIN).

snbr A number that specifies how many semaphores are to be opened by this call (FIXED BIN).

ids(x) An array of semaphore numbers; one number is returned for each semaphore that was successfully opened (FIXED BIN).

init-val The initial value (-32767 to -1) to be assigned to the semaphore.

code A success/failure code (FIXED BIN):

0 Success.

E$BPAR An invalid value was supplied for snbr.

E$IREM A file that is on a remote disk was specified in the fname parameter — remote files cannot be used as parameters to this call.

E$FUIU Either the user has all available file units opened, or that there are no available named semaphores.

E$UNOP Unopened file unit.

E$BUNT Bad file unit. (Units 1 through 127 are allowed; 127 is the COMOUTPUT file unit.)

It is also possible that code will be set to any error code that can be returned by the SRCH$$ routine.

Discussion

To open a set of named semaphores, a call must associate them with a file system object. SEM$OP will open a set of named semaphores associated with the name of a file in the current UFD of the process performing the open operation. If the process has at least read access rights to the file, it will be assigned the semaphores. Each semaphore will be initialized to 0. SEM$OU will open a set of named semaphores, associating with them a file open on a particular file unit. As before, if the process has at least read access rights to the file, it will be assigned the semaphores. Unlike SEMOP, SEMOU allows each semaphore within the set to be initialized to a nonpositive value, not less than -32767 decimal. All calls to either SEM$OP or SEM$OU which use the same file will result in the same semaphore numbers being returned.

On Rev. 19 or higher of PRIMOS, it is possible for a number of processes to have access to a set of semaphores while other processes are denied access to the same semaphores. These semaphores are called protected or named semaphores and are discussed above.

To access a named semaphore, a call must be made to SEM$OP, which grants or denies access to the semaphore. The process supplies a filename to the call. If the specified file can be accessed for read access, subject to file system and ACL protections, then the user is given access to the requested semaphores. Multiple semaphores can be opened in a single call by supplying the number of semaphores needed in the snbr parameter.

If access is granted to the semaphores, then the call will return an array of semaphore numbers in the ids parameter. One number will be returned for each semaphore requested in snbr, assuming enough semaphores exist in the system pool. A semaphore number of 0 will be returned if a semaphore could not be assigned. In addition, code will be nonzero if one or more semaphore numbers could not be assigned. The values returned in ids should be examined to determine which semaphores were opened (nonzero value returned), and which were not (0 value returned).

The semaphore numbers returned should be used in all other semaphore calls as the semaphore number parameter. SEM$OP takes a filename and returns semaphore numbers; SEM$OU takes a file unit; the rest of the calls accept only a semaphore number.

If different processes call SEM$OP or SEM$OU and specify the same filename or file unit, the same semaphore numbers will be returned to each process. This allows multiple processes of a subsystem to reference common semaphores.

If a call to the open routine specifies the same filename or unit number as a previous call to open, and a larger number of semaphores is requested, then new semaphores are acquired from the system pool to make up the difference between the number currently open (with that filename or unit number) and the number requested in the call. Other processes cannot use these newly assigned semaphores unless they explicitly open them via a call to the open routine.

When the first process opens a named semaphore, the operating system will set the value of the semaphore counter to 0 or to the number specified by SEM$OU. Subsequent opens of the semaphore do not alter the value of the counter. If a process opens the same semaphores more than once, then the same semaphore numbers will be returned for each call. No matter how many times a process opens a semaphore, it need only close that semaphore once. This removes the burden of counting to be sure that equal numbers of open and close calls are done.

Named semaphores can only be opened for files that reside on a local computer system. Attempts to open named semaphores with filenames that are on remote disks will result in failure; no semaphore numbers will be assigned and code will be set to E$IREM.

If a file that was used in a call to SEM$OP or SEM$OU is deleted or renamed while the semaphores assigned by such a call are still open, or if the disk on which that file resides is shut down, then the open semaphores will continue to be accessible to the processes that already have them open. New processes will not be given access to those semaphores, even if the disk is added again, or if a file is created with the same name as the one that was renamed or deleted. Processes that have the semaphores open can continue to use them until they are closed via a call to SEM$CL.

If a process logs out before all named semaphores have been closed, then those that are still open will be automatically closed by the operating system.

▶ SEM$NF

▶ SEM$WT

Purpose

SEM$NF releases the next process waiting on a semaphore. SEM$WT places a process in the queue for a semaphore.

Usage

```
CALL SEM$NF (snbr, code)

CALL SEM$WT (snbr, code)
```

snbr — A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or it can be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).

code — A success/failure code returned by the routine (FIXED BIN):

0	Success.
E$BPAR	Indicates that an invalid value was supplied for snbr.

E$BDAT	Indicates bad data supplied; the System Administrator should be notified.

Discussion

As explained in an earlier section, the notify and wait operations are the basic functions that can be performed on a semaphore. Notify decrements the semaphore's counter and will release the first process from the wait queue, if there are any processes waiting.

Wait increments the semaphore's counter and places the process on the semaphore's queue if the counter becomes greater than 0. Processes are queued first-in-first-out within process priority; higher priority processes are queued before those with lower priority.

▶ SEM$TS

Purpose

SEM$TS tests the counter for the number of processes waiting in the queue for a semaphore.

Usage

sval = SEM$TS (snbr, code)

sval — The current value of the specified semaphore's counter word (FIXED BIN).

snbr — A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or it can be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).

code — A success/failure code returned by the routine (FIXED BIN):

0	Success.
E$BPAR	An invalid value was supplied for <u>snbr</u>.

Discussion

This operation returns the current value of the counter, for semaphore numbered snbr in the variable sval.

▶ SEM$DR

Purpose

SEM$DR resets the specified semaphore counter to 0 (drains it).

Usage

CALL SEM$DR (snbr, code)

snbr	A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or it can be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).
code	A success/failure code returned by the routine (FIXED BIN):

0	Success.
E$BPAR	An invalid value was supplied for snbr.

Discussion

If, at the time of the SEM$DR call, the semaphore's counter is less than or equal to 0, the counter is set to 0. If, however, the counter is greater than 0, then notifies are done on the semaphore until the counter reaches 0. This causes all processes that were waiting on the semaphore to be removed from the wait queue of the semaphore.

It is possible for processes to be placed on the wait queue while this call is executing; these added processes may not be removed when the SEM$TS call returns to its caller.

▶ SEM$TN

Purpose

This operation causes the operating system to notify the specified semaphore on a periodic basis. This timer is set only for numbered semaphores.

Usage

CALL SEM$TN (snbr, int1, int2, code)

snbr — A semaphore number; it must be a number in the allowable range for numbered semaphores (0-64) (FIXED BIN).

int1 — The amount of clock time in milliseconds that will pass before the system notifies the semaphore the first time (FIXED BIN).

int2 — The amount of clock time that will pass before the semaphore is notified the second and subsequent times (FIXED BIN). If int2 is 0, then the semaphore will only be notified once - after int1 milliseconds. Specifying both int1 and int2 as 0 will remove a previous timer request from the semaphore. This is necessary when a previous SEM$TN call was made with int1 and int2 both nonzero.

If a call is made to SEM$TN which specifies a semaphore that already has an active timer request, then the values of int1 and int2 specified in the latter call will overwrite the values stored in the active timer. Note: it is possible to delay a notify caused by a timeout indefinitely by making repeated calls to SEM$TN.

code — A success/failure code returned by the routine. The values of the code are the same as those returned by SEM$WT and SEM$NF (FIXED BIN).

Discussion

The operating system maintains a limited number of timers for numbered semaphores. Currently, there are a total of 15 such timers per system.

▶ SEM$TW

Purpose

This routine allows a process to wait on the specified semaphore until it is taken off the wait queue by a notify, or until a specified amount of realtime has elapsed, whichever comes first. It is used only for named semaphores.

Usage

CALL SEM$TW (snbr, intl, code)

snbr — A semaphore number; it must be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).

intl — A time interval expressed in tenths of a second of clock time (FIXED BIN).

code — A value that indicates why the process was allowed to continue (FIXED BIN):

0	The process was notified by a call to SEM$NF.
1	The specified amount of time has elapsed and the process has not yet been notified out of the wait queue.

▶ SEM$CL

Purpose

SEM$CL releases (closes) a semaphore.

Usage

CALL SEM$CL (snbr, code)

snbr — A semaphore number; it must be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).

code — A success/failure code returned (FIXED BIN). Values are the same as for SEM$OP and SEM$OU.

Discussion

When a process no longer needs a named semaphore, it can tell the operating system that it is done with it by calling SEM$CL, to close the semaphore. After this call, the specified semaphore number cannot be used again by the process, unless that same number is reassigned by another call to SEM$OP or SEM$OU.

When a process logs out, all semaphores that were opened by that process but not explicitly closed are automatically closed by the operating system.

► SLEEP$

Purpose

SLEEP$ suspends a process for a specified interval.

Usage

CALL SLEEP$(interval)

interval	A variable containing the interval in milliseconds for which execution is to be suspended (INTEGER*4).

Discussion

Execution of the user process is suspended for the specified interval. An interval less than 0 will have no effect. A QUIT and START from the user terminal will cause immediate reexecution of the SLEEP$ call.

Note

Although the sleep interval is specified in milliseconds, SLEEP$ truncates it to an accuracy of tenths of seconds.

PART VII
Condition Handling

22 Condition Mechanism Subroutines

INTRODUCTION

This chapter 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 for programs to handle error conditions without forcing a return to command level.
- Provide support for the condition mechanism of ANSI PL/I.

When such an unusual event occurs, its corresponding on-unit (a procedure or a block of code) is executed. The subroutines described in this chapter allow the programmer to create and use on-units. These features are available to programmers using FTN, F77, PL1G, and PMA. The descriptions below use mostly PL/I terminology, with special advice for FORTRAN users.

This chapter contains a list of system-defined conditions. Because PRIMOS error handling uses conditions, the list of condition names is helpful in interpreting error messages printed by PRIMOS.

Table 22-1
Subroutines Appropriate to Various Languages

	Programming Language(1)			
Action	FTN	F77	PL1G	PMA
Create an on-unit	MKON$F	MKON$P	MKON$P(2)	MKONU$(3)
Signal a condition	SGNL$F	SGNL$F	SIGNL$	SIGNL$
Cancel (revert) an on-unit	RVON$F	RVON$F	RVONU$(4)	RVONU$
Nonlocal GOTO	PL1$NL	PL1$NL	(5)	PL1$NL
Make PL/I-compatible label	MKLB$F	MKLB$F	(5)	MKLB$F

Numbers in parenthesis refer to the following notes.

1. The CPL language, not shown in this table, also supports the condition mechanism, but without the use of these subroutine calls. See EXAMPLES OF PROGRAMS below.

2. MKON$P required for programmer-named condition. Several predefined conditions are supported by the language's ON statement. It is also possible to use MKONU$ instead of MKON$P. See MKONU$ under CONDITION MECHANISM SUBROUTINES, later in this chapter.

3. The user must provide extended stack area, and, while the condition handler is active, must not modify the character-varying variable which holds the condition name.

4. Or use the language-supplied REVERT statement.

5. Supported directly by the programming language.

CREATING AND USING ON-UNITS

Condition handlers are called on-units. They 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 following subroutines:

MKONU$

MKON$F

MKON$P

The use of these subroutines is the only way to create an on-unit in a non-PL/I environment. See Table 22-1 to determine which subroutine to use.

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

Again, use Table 22-1 to select the proper subroutine.

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 chapter.

It is extremely important that any on-unit procedure take at least one argument.

On-unit Actions

An on-unit has several options on action it may take. An on-unit may:

- Perform application-specific tasks (such as closing or updating files).
- Repair cause of condition and resume execution.
- Decide that normal flow can be interrupted and program reentered at a "known point" by performing a nonlocal GOTO to some previously defined label.
- Signal another condition.
- Transfer process to command level.
- Continue search for more on-units.
- 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 a 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.

This chapter documents subroutines in PL/I notation. FORTRAN users may convert between PL/I and FORTRAN data types by using Table 22-2.

Table 22-2
Conversion of PL/I to FORTRAN Data Types

PL/I	FORTRAN
char(n) var	INTEGER(((n+1)/2)+1)
char(n)	INTEGER((n+1)/2)
fixed bin(15)	INTEGER*2
fixed bin(31)	INTEGER*4
label	REAL*8
entry variable	REAL*8
ptr options (short)	INTEGER*4
bit(n)	INTEGER*2 (1<=n<=16)

The PL/I interfaces use the PL/I data type "character(*) varying". This data type is not available in FORTRAN, but 1977 ANSI FORTRAN (F77) includes a data type "character*n" which is the equivalent of PL/I "character(n) nonvarying". Interfaces are provided which use the nonvarying 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$'/
```

On-units must be carefully designed not to require reentrancy, which is not supported by FORTRAN. See how I/O must be handled in EXAMPLES OF PROGRAMS, below.

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 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. Otherwise, if an ANY$ on-unit exists, the ANY$ on-unit is selected.

User programs should avoid the use of the ANY$ on-unit. A user's 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 indicates 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 the switch be turned on. This switch is cleared before each on-unit (except ANY$) is invoked. See the discussion of the continue switch at cflags.continue_sw under DATA STRUCTURE FORMATS later in this chapter.

EXAMPLES OF PROGRAMS

Below are sample programs in FORTRAN 66 (FTN), FORTRAN 77 (F77), PL/I Subset G (PL1G), and CPL which use an on-unit to trap the QUIT$ condition. The programs are similar, but not identical, in operation.

Note

In both FORTRAN examples (FTN and F77), the on-unit must avoid using standard FORTRAN I/O, and instead uses TNOU. The condition has arisen in the middle of FORTRAN input, and since FORTRAN I/O is not reentrant, use of FORTRAN I/O by the on-unit would destroy the environment to which it eventually returns. PL1G supports reentrancy, and does not require this precaution.

FORTRAN Example

```
C Program to demonstrate on-unit in FTN
C
      EXTERNAL CATCH
      INTEGER*2 BREAK(3), BREAKL, I
      DATA BREAK/'QUIT$'/
      BREAKL = 5
      CALL MKON$F(BREAK, BREAKL, CATCH)
      WRITE(1,300)
300   FORMAT('Please enter an integer, then RETURN.')
100   CONTINUE
```

```
      READ(1,200) I
200   FORMAT(I8)
      WRITE(1,330)
330   FORMAT('Again, 0 to exit, BREAK to test on-unit.')
      IF (I .NE. 0) GOTO 100
      STOP
      END
C
      SUBROUTINE CATCH(PNTR)
      INTEGER*4 PNTR
      CALL TNOU('We caught a quit!',17)
      PAUSE 1
      CALL TNOU('You''re back into the input loop again.',38)
      RETURN
      END
```

FORTRAN 77 Example

```
C Program to demonstrate on-unit in F77
C
      external catchit
      integer*2 break_length
      character*5 break/'QUIT$'/
      break_length = 5
      call mkon$p(break,break_length,catchit)
      print*, 'Please enter an integer, then RETURN.'
100   continue
      read(1,*) i
      print*, 'Again, 0 to exit, BREAK to test on-unit.'
      if (i.ne.0) goto 100
      end
      subroutine catchit(pntr)
      integer*4 pntr
      call tnou('We caught a quit!',ints(17))
      pause 1
      call tnou('You''re back into the input loop again.',ints(38))
      return
      end
```

PL/I Subset G Examples

```
/* Program to demonstrate on-unit in PL1G */

ex_pl1g: procedure options (main);
   dcl mkon$p entry(char(*), fixed bin, entry);
   dcl (break_length, i) fixed bin(15);
   dcl (break) character(5) static initial('QUIT$');
   break_length = 5;
   call mkon$p(break, break_length, catchit);
   put skip list ('Please enter an integer, then RETURN.');
   get list (i);
   do while (i ^= 0);
      put skip list ('Again, 0 to exit, BREAK to test on-unit.');
      get list (i);
   end;
   stop;

   catchit: proc (pntr);
      dcl pntr pointer;
      put skip list ('We caught a quit!');
      put skip list('You''re back into the input loop again.');
      return;
   end;
end;
```

```
 /* Modified program to demonstrate on-unit in PL1G */
 /* Shows use of MKONU$ (instead of MKON$P) */

ex_pl1g: procedure options (main);
   declare mkonu$ entry(character(32) varying, entry)
                  options(shortcall(20));
   declare (break) character(32) static initial('QUIT$') varying;
   declare i fixed binary(15);
   call mkonu$(break, catchit);
   put skip list ('Please enter an integer, then RETURN.');
   get list (i);
   do while (i ^= 0);
      put skip list ('Again, 0 to exit, BREAK to test on-unit.');
      get list (i);
   end;
   stop;

   catchit: procedure (pntr);
      declare pntr pointer;
      put skip list ('We caught a quit!');
      put skip list('You''re back into the input loop again.');
      return;
   end;
end;
```

CPL Example

```
/* Program to demonstrate on-unit in CPL.
   Note that CPL cannot call a make-on-unit
   subroutine.  Instead, we show the use of
   the ON statement provided by CPL.
 */
&on QUIT$ &routine catchit
type 'Please enter an integer, then RETURN.'
&set_var i := [response '']
```

```
&do &while %i% ^= 0
   type 'Again, 0 to exit, BREAK to test on-unit.'
   &set_var i := [response '']
&end
&stop

&routine catchit
type 'We caught a quit!'
type 'You''re back into the input loop again.'
&return
```

ADDITIONAL EXAMPLE PROGRAMS

Several programs presented below show strategies for using the condition mechanism. The examples include:

- CPL programs to do on-unit handling for a program which does not itself use on-units.
- A FORTRAN 77 (F77) program to show reentering a program with the PRIMOS REN command. The program also shows the use of nonlocal GOTO.
- A FORTRAN 66 (FTN) program handling QUIT$ and showing nonlocal GOTO.
- A PL/I Subset G (PL1G) program handling end of file.
- A FORTRAN 66 program which demonstrates the CLEANUP$ condition, which is raised during processing of a nonlocal GOTO.

Two Protecting Programs in CPL

Below are two programs each of which protects a FORTRAN program called SQRT against being interrupted by the BREAK (or CONTROL-P) key. They demonstrate both a simple and a more sophisticated means by which programs can avoid having to use the condition mechanism subroutines. When the language in which a program was written does not support on-units, or when condition handling is to be added as an afterthought, CPL can sometimes be used to handle conditions.

```
/* PROTECT.CPL
/* Trap the BREAK key with an on-unit in CPL.
/*
&ON QUIT$ &ROUTINE BREAK_HANDLER
&DATA SEG SQRT
  &TTY
&END
&RETURN
```

```
    &ROUTINE BREAK_HANDLER
      TYPE
      TYPE
      TYPE You have typed the break key.
      &SET_VAR EXIT_FLAG := ~
        [QUERY 'Do you wish to exit from the program']
      &IF ^  ~
      &THEN ~
        TYPE Continuing program.
      &ELSE ~
        &DO
          TYPE Exiting program.
          &STOP
        &END
      &RETURN
```

The program PROTECT2.CPL can better handle the user's typing several BREAKs in a row.

```
    /* PROTECT2.CPL
    /* Trap the BREAK key with an on-unit in CPL.
    /* Do not allow multiple breaks.
    /*
    &ON QUIT$ &ROUTINE BREAK_HANDLER
    &DATA SEG SQRT
      &TTY
    &END
    &RETURN

    &ROUTINE BREAK_HANDLER
      &ON QUIT$ &ROUTINE DUMMY_HANDLER
      TYPE
      TYPE
      TYPE You have typed the break key.
      &LABEL ALTERNATE_ENTRY
      &SET_VAR EXIT_FLAG := ~
        [QUERY 'Do you wish to exit from the program']
      &IF ^  ~
      &THEN ~
        TYPE Continuing program.
      &ELSE ~
        &DO
          TYPE Exiting program.
          &STOP
        &END
      &RETURN

    &ROUTINE DUMMY_HANDLER
      TYPE
      TYPE Please answer the question!
      &GOTO ALTERNATE_ENTRY
      &RETURN
```

Here is the FORTRAN source for the SQRT program invoked by PROTECT and PROTECT2.

```
C  SQRT.FTN
C
C  This is a small interactive FORTRAN program which is to be
C  protected from BREAKs (the QUIT$ condition) by an enveloping
C  program written in CPL.
C
      REAL INVAL, OUTVAL
C
1000  WRITE (1, 1005)
1005  FORMAT (/, 'WHAT IS THE NUMBER:')
      READ (1, 1010) INVAL
1010  FORMAT (F5.0)
      IF (INVAL .EQ. 0.) GOTO 9999
      OUTVAL = SQRT (INVAL)
      WRITE (1, 1020) INVAL, OUTVAL
1020  FORMAT ('THE SQUARE ROOT OF ', F5.0, ' IS ', F5.2)
      GOTO 1000
C
9999  WRITE (1, 9000)
9000  FORMAT (/ , 'END OF PROGRAM')
      CALL EXIT
      END
```

The REENTER$ Condition from F77

```
C  REENTER.F77
C
C  This program creates an on-unit for the REENTER$ condition.
C  If the user breaks out of the program during its operation, and
C  then reenters it through the PRIMOS REN command, the on-unit
C  will be invoked to start the program from the proper place.
C
      EXTERNAL RENHDLR
      EXTERNAL MKON$P
      EXTERNAL MKLB$F
C
      CHARACTER*8 CONDITION_NAME/'REENTER$'/
      CHARACTER*80 CHAR_STRING
      REAL*8 REENTRY_POINT
      INTEGER*2 INDEX, CONDITION_LENGTH/8/
C
      COMMON /REENTRY/ REENTRY_POINT
C
C  The "$1000" on the next line refers to statement 1000
      CALL MKLB$F ($1000, REENTRY_POINT)
      CALL MKON$P (CONDITION_NAME, CONDITION_LENGTH, RENHDLR)
C
```

```
1000 WRITE (1, 1010)
1010 FORMAT ('Enter a character string:')
     READ (1, 1020) CHAR_STRING
1020 FORMAT (A80)
C
     DO 9999 INDEX = 1, 500
     WRITE (1, 1030) CHAR_STRING
1030 FORMAT (A80)
9999 CONTINUE
     END
C
C
     SUBROUTINE RENHDLR (CP)
C
     INTEGER*4 CP
C
     EXTERNAL PL1$NL
     COMMON /REENTRY/ REENTRY_POINT
     WRITE (1, 1010)
1010 FORMAT ('** Reentering subsystem **')
     CALL PL1$NL (REENTRY_POINT)
     RETURN
     END
```

Handling QUIT$ from FTN

```
C  PROSQRT.FTN
C
C  This program creates an on unit for the BREAK key.  The on-unit
C  prevents BREAK from exiting the program, and instructs the user
C  how to exit.
C
C  In FTN the on-unit must be declared as an external routine.
C
      EXTERNAL BKHNDL
C
      REAL INVAL, OUTVAL
      REAL*8 BRKRTN
C
      COMMON /BRKLBL/ BRKRTN
C
      CALL MKON$F ('QUIT$', 5, BKHNDL)
C  The "$1000" in the next line refers to statement 1000
      CALL MKLB$F ($1000, BRKRTN)
1000  WRITE (1, 1005)
1005  FORMAT (/, 'WHAT IS THE NUMBER:')
      READ (1, 1010) INVAL
1010  FORMAT (F5.0)
      IF (INVAL .EQ. 0.) GOTO 9999
      OUTVAL = SQRT (INVAL)
      WRITE (1, 1020) INVAL, OUTVAL
1020  FORMAT ('THE SQUARE ROOT OF ', F5.0, ' IS ', F5.2)
```

```
      GOTO 1000
C
9999  WRITE (1, 9000)
9000  FORMAT (/ , 'END OF PROGRAM')
      CALL EXIT
      END


C
C  This subroutine handles the QUIT$ condition when it is raised.
C
C  Ordinarily, it would be incorrect to use FORTRAN I/O from inside
C  this on-unit, because FORTRAN is not reentrant, and we would
C  be disturbing the keyboard I/O which was in progress when QUIT$
C  was raised.  In this case, however, we use a nonlocal GOTO to
C  return to statement 1000 of the main program, and never return
C  to the I/O which was in progress.
C
      SUBROUTINE BKHNDL (CP)
C
      INTEGER*4 CP
      REAL*8 BRKRTN
      COMMON /BRKLBL/ BRKRTN
      WRITE (1, 1000)
1000  FORMAT ('YOU MUST TYPE ZERO TO EXIT THIS PROGRAM!')
      CALL PL1$NL (BRKRTN)
      RETURN
      END
```

Handling End of File from PL1G

```
/*  EOF.PL1G  */

/*  This program creates on-units for both the ENDFILE and QUIT$
    conditions.  The on-unit for the end-of-file condition is
    set up by PL/I's "ON" statement, while the on-unit for quits
    is set up by calling MKON$P.  The on-unit for quits closes
    all files and exits the program.
 */
EXAMPLE: PROCEDURE OPTIONS(MAIN);

    DCL EMPLOYEE_NO FIXED DECIMAL(5);
    DCL (GROSS_PAY, HOURLY_RATE) FIXED DECIMAL(5,2);
    DCL HOURS_WORKED FIXED DECIMAL(2);
    DCL FIXED DECIMAL(5,2);
    DCL NUMBER_OF_EMPLOYEES FIXED BIN(15);
    DCL (REPORT, DATAFILE) FILE;
    DCL CONDITION_NAME CHAR (5) STATIC INITIAL ('QUIT$');
    DCL MKON$P ENTRY (CHAR (5), FIXED BIN, ENTRY);

    BREAK_HANDLER: PROC (CP);
        DCL CP PTR;
        PUT SKIP LIST ('** Aborting program **');
        CLOSE FILE (DATAFILE);
        CLOSE FILE (REPORT);
        GOTO ABORT_PROGRAM;
    END;

    ON ENDFILE (DATAFILE)
    BEGIN;
        PUT SKIP LIST ('** End of File Encountered **');
        GOTO END_FILE;
    END;

    CALL MKON$P (CONDITION_NAME, 5, BREAK_HANDLER);
    OPEN FILE (DATAFILE) TITLE ('DATAFILE') STREAM INPUT;
    OPEN FILE (REPORT) TITLE  ('REPORT')  STREAM OUTPUT;
    NUMBER_OF_EMPLOYEES = 0;

    DO WHILE ('1'B);
        GET FILE (DATAFILE)
            LIST (EMPLOYEE_NO, HOURLY_RATE, HOURS_WORKED);
        NUMBER_OF_EMPLOYEES = NUMBER_OF_EMPLOYEES + 1;
        GROSS_PAY = HOURS_WORKED * HOURLY_RATE;
        PUT FILE (REPORT)
         LIST (EMPLOYEE_NO, HOURLY_RATE,
               HOURS_WORKED, GROSS_PAY);
        PUT FILE (REPORT) SKIP;
    END;

    END_FILE:
    PUT FILE(REPORT) LIST(NUMBER_OF_EMPLOYEES) SKIP(3);

    ABORT_PROGRAM:
END EXAMPLE;
```

A CLEANUP$ On-unit from FTN

The following programs demonstrate the QUIT$ and CLEANUP$ on-units. When the BREAK key is typed, a nonlocal GOTO is executed, which causes CLEANUP$ to be raised in the routine SUBA.

```
C  CLEANUP.FTN
C
C  This program creates on-units for the QUIT$ and CLEANUP$
C  conditions.
C
      EXTERNAL BKHNDL
C
      REAL*8 BRKRTN
      COMMON /BRKLBL/ BRKRTN
C
      CALL MKON$F ('QUIT$', 5, BKHNDL)
      CALL MKLB$F ($1000, BRKRTN)
1000  WRITE (1,1010)
```

```
1010  FORMAT (/, 'In the routine: MAIN')
      CALL SUBA
      CALL EXIT
      END
C
      SUBROUTINE SUBA
      EXTERNAL ACLUP
      WRITE (1, 1000)
1000  FORMAT ('In the routine: SUBA')
      CALL MKON$F ('CLEANUP$', 8, ACLUP)
      CALL SUBB
      RETURN
      END
C
      SUBROUTINE SUBB
      INTEGER DUMMY
      WRITE (1,1000)
1000  FORMAT ('In the routine: SUBB')
      WRITE (1, 1010)
1010  FORMAT ('Type RETURN to exit, BREAK to test on-units')
      READ (1, 1020) DUMMY
1020  FORMAT (A2)
      RETURN
      END


C  HDLRS.FTN
C
C  On-units for the module CLEANUP.FTN
C
C  The routine ACLUP is invoked when a non-local goto is
C  aborting SUBA.
C
      SUBROUTINE ACLUP (CP)
      INTEGER*4 CP, I
      WRITE (1, 1000)
1000  FORMAT ('In the cleanup routine: ACLUP')
      DO 1010 I = 1, 50000
1010  CONTINUE
      RETURN
      END
C
C  The routine BKHNDL is invoked when the QUIT$ condition is
C  raised by the user hitting the BREAK key.
C
      SUBROUTINE BKHNDL (CP)
      INTEGER*4 CP
      REAL*8 BKRTN
      COMMON /BRKLBL/  BRKRTN
      WRITE (1, 1000)
1000  FORMAT ('In the routine: BKHNDL')
      CALL PL1$NL (BRKRTN)
      RETURN
      END
```

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. (Protection rings are described in the System Architecture Reference Guide.) 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.

CONDITION MECHANISM SUBROUTINES

The user-level subroutines for the condition mechanism are described below in alphabetical order.

▶ CNSIG$

Purpose

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-to-signal switch, cfh.cflags.continue_sw, is set in the most recent condition frame.

Usage

```
DCL CNSIG$ ENTRY (FIXED BIN);

CALL CNSIG$ (status);
```

status	Standard system error code: will be nonzero only if there was no condition frame found in the stack.

Discussion

The continue-to-signal 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

Purpose

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.

Usage

```
INTEGER*2 stmt
REAL*8 label
CALL MKLB$F (stmt, 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 returned by call to MKLB$F.

▶ MKON$F

Purpose

MKON$F creates an on-unit for a specific condition and is intended for the FORTRAN user.

Usage

The FORTRAN usage is:

```
EXTERNAL unit
INTEGER*2 cname(—), cnamel
CALL MKON$F (cname, cnamel, unit)
```

cname — Array containing name of condition for which on-unit is to be created.

cnamel — Length (in characters) of cname.

unit — Your external subroutine which is to be the on-unit handler. Your subroutine must take an argument, since the PRIMOS condition mechanism calls your subroutine as follows:

```
INTEGER*4 CP
CALL UNIT (CP)
```

where CP is a pointer to the condition frame header (CFH) that describes the condition.

Discussion

FORTRAN cannot directly access the CFH through CP. A subroutine written in PL1G or PMA would be required to pass the desired CFH information.

Cname and cnamel may be overwritten by the caller once MKON$F has returned, since they are copied into a stack frame extension.

> **Caution**
>
> MKON$F cannot be called from FORTRAN 77. FORTRAN 77 requires MKON$P.

▶ MKON$P

Purpose

MKON$P creates an on-unit for a given condition. It may be used in FORTRAN 77 and PL1G programs.

Usage

The PL1G usage is:

```
DCL MKON$P ENTRY (CHAR(*), FIXED BIN, ENTRY);

CALL MKON$P (condname, namelen, handler);
```

condname The name of the condition for which an on-unit is desired. The name should not contain any blanks (input).

namelen The length of condname, in characters (input).

handler The internal or external entry (subroutine) value which is to be invoked as the on-unit. If the value is an internal procedure, it must be immediately contained in the block calling MKON$P (input). The subroutine must take at least one argument.

An on-unit for the specified named condition is created for the calling block. If the block already has an on-unit for that condition, the on-unit is redefined.

The F77 usage is:

```
EXTERNAL handler
INTEGER*2 namelen
CHARACTER*namelen name/'condname'/

CALL MKON$P(name, namelen, handler)
```

condname The name of the condition for which an on-unit is desired. The name should not contain any blanks (input).

namelen The length of condname, in characters (input).

name A variable to hold condname. Its value should not be altered while the condition is active.

handler The name of the external subroutine which is to become the on-unit. This subroutine must take at least one argument.

Discussion

Caution

MKON$P cannot be called from FORTRAN (FTN). FORTRAN requires MKON$F.

▶ MKONU$

Purpose

MKONU$ creates an on-unit for a specific condition or creates a default on-unit for the ANY$ condition. MKONU$ can be called only from PMA and PL1G. PL1G programmers may use either MKON$P or MKONU$. From PL1G the declaration OPTIONS (SHORTCALL) is required for MKONU$. See below.

Usage

```
DCL MKONU$ ENTRY OPTIONS(SHORTCALL stack_increase) (CHAR(*)VAR, ENTRY);

CALL MKONU$ (condition_name, handler);
```

stack_increase — Additional space needed for the calling procedure's temporary storage. OPTIONS(SHORTCALL) provides 8 words of stack by default. MKONU$ requires 28 words of stack, and thus requires stack_increase of 20. If the stack size is not large enough, the return from MKONU$ will cause unpredictable results.

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 overwritten.

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.) The handler must take at least one argument.

Discussion

The stack frame of the caller is lengthened, 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.

OPTIONS(SHORTCALL) causes the PMA instruction JSXB to be used instead of the PCL instruction. PCL generates a new stack. JSXB does not, and is faster, but requires that there be sufficient space on the caller's stack. MKONU$ is the only Rev 18 or 19 system subroutine that can (and must) be declared this way.

▶ PL1$NL

Purpose

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.

Usage

REAL*8 label
CALL PL1$NL (label)

label	PL/I - compatible label value.

▶ RVON$F

Purpose

RVON$F disables an on-unit for a specific condition. Its effect is identical to RVONU$ but is designed for the FORTRAN user. RVON$F is used from FORTRAN and FORTRAN 77.

Usage

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.

Discussion

There is no effect if an on-unit does not exist for the named condition, or if the on-unit has already been disabled.

▶ RVONU$

Purpose

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 reinstated only by another call to MKONU$ or MKON$F. A call to RVONU$ affects only on-units within its own activation. RVONU$ is used from PL1G and PMA programs.

Usage

```
DCL     RVONU$ ENTRY (CHAR(*) VAR);

CALL    RVONU$ (condition_name);
```

condition_name — Name of condition for which the on-unit is to be disabled.

Discussion

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.

▶ SGNL$F

Purpose

SGNL$F signals a specific condition and supplies optional auxiliary information. SGNL$F is the FORTRAN equivalent of SIGNL$. It is used from FORTRAN and FORTRAN 77 programs.

Usage

```
INTEGER*2 cname(—), cnamel, mslen, infoln, flags
INTEGER*4 msptr, infopt
CALL SGNL$F (cname, cnamel, msptr, mslen, infopt, infoln, flags)
```

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 (octal).

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	Meaning
1	If =1, on-unit may return.
2	If =1, on-unit may return without taking action.
3	If =1, call is result of crawlout. This bit should never be set by the user.
4	If =1, signal PLIO condition. User program should not set.
5-16	Must be 0.

► SIGNL$

Purpose

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$) on-unit. SIGNL$ is used from PL1G and PMA programs.

Usage

```
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 A 16-bit word that defines action to be taken:

```
DCL 1 action,
     2 return_ok bit(1),
     2 inaction_ok bit(1),
     2 crawlout bit(1),
     2 specifier bit(1),
     2 mbz bit(12);
```

return_ok If = '1'b, on-unit is to be allowed to return.

inaction_ok If = '1'b, on-unit may return without taking corrective action and still expect "defined" results. (return_ok must also be '1'b.)

crawlout If = '1'b, call to SIGNL$ is result of crawlout. Should never be set by user.

specifier If = '1'b, signals PL/I I/O(PLIO) condition. User program should not use.

mbz Must be zero.

SYSTEM-DEFINED CONDITIONS

The following are the standard system-defined conditions. The meaning of each condition is given, followed by a description of the information available in the condition frame header structure produced by that 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 data structures used by the condition mechanism, including the Condition Frame Header (CFH), the Stack Frame Header (SFH), the Fault Frame Header (FFH), and the on-unit descriptor block, are discussed later in this chapter under DATA STRUCTURE FORMATS.

In the descriptions below, software means that the machine state frame pointed to by cfh.ms_ptr is a condition frame header, and hardware means that this frame is a fault frame header. The notations "ffh." and "cfh." below refer to the fault frame header or condition frame header that is pointed to by ffh.ms_ptr or cfh.ms_ptr. The information structures referred to below are pointed to by cfh.info_ptr.

Unless otherwise noted below, the system default on-unit for each condition prints an appropriate diagnostic message on the user's terminal, terminates program execution, and returns to PRIMOS command level.

▶ ACCESS_VIOLATION$

(hardware, returnable)

The process has attempted to perform a CPU instruction which has violated the access control rules of the processor. No information is readily available to differentiate between write violation, read violation, execute violation, and gate violation.

ffh.fault-type	Value '44'b3.
ffh.fault_addr	Contains the virtual address whose access is improper.

ffh.ret_pb Points to the instruction causing the violation.

No information structure is available.

▶ ANY$

(pseudo-condition)

An activation's on-unit for ANY$ is invoked if that activation does not have a specific on-unit for the condition that was raised. The condition frame header for the condition ANY$ will describe the original condition directly; there is no separate condition frame header for the condition ANY$ unless ANY$ has been explicitly raised by a call to SIGNL$ (not a recommended practice).

▶ AREA

(software, not returnable)

This condition is raised when a storage area has been damaged, or when the target area for an attempted copy from one area to another was too small. (Generally raised by full PL/I only. Not available through PL1G.)

▶ ARITH$

(hardware, returnable)

The process has encountered an arithmetic exception fault.

ffh.fault_type Value '50'b3.

ffh.fault_code Hardware-defined exception code which partially identifies the cause of the fault.

ffh.ret_pb Points to the next instruction to be executed upon return. There is no way in general to obtain a pointer to the faulting instruction.

No information structure is available.

The static-mode default on-unit for this condition will simulate Prime 300 fault handling for arithmetic exception if the appropriate word of segment '4000 is nonzero. (See the System Architecture Reference Guide for the exact location.) If a static-mode program is not in execution when the fault occurs, or if the Prime 300 vector word is 0, the standard default handler for this condition will resignal the appropriate arithmetic condition (size, fixedoverflow, etc.) with the appropriate information structure.

▶ BAD_NONLOCAL_GOTO$

(software, not returnable)

The nonlocal GOTO processor has been asked to transfer control to a label whose display (stack) pointer is invalid, or whose target activation has already been cleaned up. There is also a possibility that the user's stack may have been overwritten.

Information Structure:

```
DCL 1 info based,
       2 target_label label,
       2 ptr_to_nlg_call ptr,
       2 caller_sb ptr;
```

info.target_label — Label to which the nonlocal GOTO was attempted.

info.ptr_to_nlg_call — Pointer to the call to PL1$NL that requested this nonlocal GOTO.

info.caller_sb — Pointer to the activation (stack frame) requesting this nonlocal GOTO.

▶ BAD_PASSWORD$

(software, not returnable)

This condition is raised by the ATCH$$ primitive when attempting to attach with an incorrect password to a directory requiring a password. This condition is signalled nonreturnable in order to increase the work function of machine-aided password penetration.

No information structure is available.

▶ CLEANUP$

(software, returnable)

The nonlocal GOTO processor (UNWIND_) is in the process of invoking on-units for the condition CLEANUP$ in each activation on the stack, 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.

No information structure is available.

▶ COMI_EOF$

(software, returnable)

End of file occurred on the command input file.

The default on-unit prints a diagnostic message and returns to the point of interrupt.

▶ CONVERSION

(software, returnable)

This condition is raised when the source data for a data-type conversion contains one or more characters that are invalid for the target type. For example, nonnumeric characters appear in a character string which is to be converted to integer.

Information Structure: Standard PL/I information structure.

▶ ENDFILE (file)

(software, returnable)

This condition is raised when an end of file is encountered while reading a PL/I file with PL/I I/O statements. The value of the ONFILE() built-in function identifies the file involved.

The standard PL/I condition information structure is provided. The value of info.oncode_value is undefined, and info.file_ptr identifies the file on which end of file occurred.

The default on-unit for this condition prints a diagnostic and then resignals the ERROR condition with an info.oncode_value of 1044.

▶ ENDPAGE (file)

(software, returnable)

This condition is raised when end of page is encountered while writing a PL/I file using PL/I I/O statements. The value of the ONFILE() built-in function identifies the file on which the end of page was encountered.

The standard PL/I condition information structure is provided. The value of info.oncode_value is undefined; info.file_ptr identifies the file in question.

The default on-unit for this condition performs a PUT SKIP on the file, and then returns.

▶ ERROR

(software, varies)

This condition is a catch-all error condition defined in PL/I. The default on-unit for most PL/I-defined conditions (such as KEY) results in the ERROR condition being resignalled. Hence, the programmer has the choice of handling a more- or less-specific case of the condition.

▶ ERRRTN$

(software, not returnable)

A nonring-0 call to the ring-0 entry ERRRTN was made, as the result of an ERRRTN SVC or a call to ERRPR$ with certain values of the key.

No information structure is available.

The default on-unit for this condition simulates a call to EXIT; hence, this condition should be signalled only while executing in a static-mode program.

EXIT$

(software, returnable)

The process has made a call to the EXIT primitive, via a direct call or an EXIT SVC. This condition should not be handled by user programs, since it is used by certain PRIMOS software to monitor the execution of static-mode programs.

No information structure is available.

The default on-unit for this condition simply returns.

FINISH

(software, returnable)

This condition is signalled before process termination. It closes any open files and returns to the point at which the condition was signalled. It is not signalled if the process is prematurely exhausted or destroyed. (Generally raised by full PL/I only. Not available through PL1G.)

The default on-unit simply returns.

FIXEDOVERFLOW

(hardware, not returnable)

This condition is detected by hardware and is raised when a fixed-point decimal or binary result is too large to fit into the hardware register or decimal field.

The standard PL/I condition information structure is provided.

ILLEGAL_INST$

(hardware, returnable)

The process has attempted to execute an illegal instruction.

ffh.fault_type	Value '40'b3.
ffh.ret_pb	Points at the faulting instruction.

No information structure is available.

▶ ILLEGAL_ONUNIT_RETURN$

(software, not returnable)

An on-unit for some condition has attempted to return, when that has been disallowed by the procedure that raised the condition.

Information Structure: The standard-format condition frame header that describes the condition whose on-unit has illegally attempted to return.

▶ ILLEGAL_SEGNO$

(hardware, returnable)

The process has referenced a virtual address whose segment number is out of bounds.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	The virtual address that is in error.

No information structure is available.

▶ KEY (file)

(software, returnable)

The KEY condition is raised when reading or writing a keyed PL/I file with PL/I I/O statements, and the supplied key does not exist (READ) or already exists (WRITE). The value of the ONFILE() built-in function identifies the file in question; the value of the ONKEY() built-in function contains the key in error.

Information Structure: The standard PL/I condition information structure. The value of info.oncode_value is undefined; the value of info.file_ptr identifies the file in question.

The default on-unit prints a diagnostic and resignals the ERROR condition, with an info.oncode_value of 1045.

▶ LINKAGE_FAULT$

(hardware, returnable)

The process has referenced through an indirect pointer (IP) which is a valid unsnapped dynamic link, but the desired entry point could not be found in any of the dynamic link tables.

ffh.fault_type   Value '64'b3.

ffh.fault_addr   Points to the faulting indirect pointer.

ffh.ret_pb   Points to the faulting instruction.

Information Structure:

```
DCL 1 info based,
      2 entry_name char(32) var;
```

info.entry_name Name of the entry point that could not be found.

▶ LISTENER_ORDER$

(software, varies)

This condition is used internally by the command loop to manage its recursion. Users should never make on-units for this condition, and user default on-units (ANY$) should always pass this condition on by returning.

▶ LOGOUT$

(software, returnable)

This condition is raised when a user or the operator is trying to force log out a process.

Information Structure:

```
DCL 1   logout_info
        2  reason fixed     /* reason for logout;
                               codes available in PRIMOS source */
```

The default on-unit logs out the process. When LOGOUT$B is signalled, the intercepting process has between one and two minutes to do its cleanup before being force-logged out.

NAME

(software, returnable)

This condition occurs only during data-directed input. It occurs when stream assignment in a GET statement is read whose variable does not match the variable name in the data list. After execution of the on-unit, the process returns to the data-directed input as if the "bad" input were processed. (Generally raised by full PL/I only. Not available through PL1G.)

NO_AVAIL_SEGS$

(hardware, returnable)

The process has referenced a virtual address that refers to a segment that has not yet been created. At the moment, 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, with some possibility of success if this or some other process has in the meantime deleted a segment.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	Virtual address that is causing the attempted segment creation.

No information structure is available.

NONLOCAL_GOTO$

(software, returnable)

This condition is signalled by the PL/I nonlocal GOTO processor PL1$NL just prior to setting up the stack unwind (and hence prior to the invocation of any CLEANUP$ on-units). This condition exists to enable certain overseer software (such as the debugger) to be informed that the nonlocal GOTO is occurring. The default handler for this condition simply returns. When a procedure handling this condition wishes to let the nonlocal GOTO occur, it should simply return (without continue-to-signal set).

Information Structure: Same as for the BAD_NONLOCAL_GOTO$ condition.

▶ NPX_SLAVE_SIGNALED$

(software, not returnable)

A condition has been raised in your NPX slave running on some remote system. The following message is printed:

```
Condition signalled in NPX slave on nodename
ERROR: Condition "condition name" raised at segment no./word no.
```

Information Structure:

```
DCL 1  npx_slave_info
       2  node fixed,     /* npx node number on which
                             slave is running */
       2  orig_condition char (32) var, /* condition
                                    raised in slave */
       2  orig_info_data (129) fixed;  /* info
                                    structure from slave */
```

When the slave detects a signalled condition, it transmits to the master, which signals the condition NPX_SLAVE_SIGNALED$. Its result is the printout of the message shown above. The slave transmits to the master almost all types of conditions signalled except the following:

EXIT$

FINISH

LINKAGE_FAULT$

NONLOCAL_GOTO$

REENTER$

STRINGSIZE

These conditions are handled differently by slave's on-unit. They are returned without transmitting to the master, that is, the master side will not get the condition NPX_SLAVE_SIGNALED$.

▶ NULL_POINTER$

(hardware, returnable)

The process has referenced 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 employ the single value '7777/0 for the null pointer.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	Null pointer through which a reference was made.

No information structure is available.

The default on-unit for this condition resignals the ERROR condition with the appropriate information structure.

▶ OUT_OF_BOUNDS$

(hardware, returnable)

The process has referenced a page of some segment that has been defined as not referencible in any ring (i.e. no main memory or backing storage is allocated for that page, and allocation is not permitted).

ffh.fault_type	Value '10'b3.
ffh.ret_pb	Points at the faulting instruction.
ffh.fault_addr	The offending virtual address.

No information structure is available.

▶ OVERFLOW

(hardware, not returnable)

This condition is raised when the result of a floating-point binary calculation is too large for representation. It may occur within a register or as a store exception. The default on-unit prints a message and signals the ERROR condition. User on-units may not return to the point of interrupt. However, if the default on-unit is invoked, and if the user types START, the register or memory location affected will be set to the largest possible single-precision floating-point number, and calculation will continue.

▶ PAGE_FAULT_ERR$

(hardware, returnable)

The process has encountered a page fault referencing a valid virtual address, but due to a disk error, the page control mechanism has not been able to load the page into main memory. If the on-unit for this condition returns, the reference will be retried, and there is some likelihood that the disk read will succeed and the reference thus be completed.

ffh.fault_type	Value '10'b3.
ffh.ret_pb	Points at the faulting instruction.
ffh.fault_addr	Virtual address, the page for which cannot be retrieved.

No information structure is available.

▶ PAUSE$

(software, returnable)

The process has executed a PAUSE statement in a FORTRAN program. This condition should not be handled by user programs since it is used by Prime software to ensure the proper operation of the FORTRAN PAUSE statement.

No information structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

▶ PH_LOGO$

(software, returnable)

This condition is raised when a phantom which you spawned is logging out.

No information structure is directly available. Use the subroutine LON$R, described elsewhere in this book.

▶ POINTER_FAULT$

(hardware, returnable)

The process has referenced through an indirect pointer (IP) whose fault bit is on, but that pointer did not appear to be a valid unsnapped dynamic link.

> **Note**
>
> This error condition is frequently caused by making a subroutine call with too few arguments. The condition is raised when the called subroutine attempts to access one of its arguments through a faulted pointer.

ffh.fault_type   Value '64'b3.

ffh.fault_addr   Points to the faulting indirect pointer.

ffh.ret_pb   Points to the faulting instruction.

No information structure is available.

▶ QUIT$

(hardware, software, returnable)

The user has actuated QUIT (BREAK key or CONTROL-P) on the terminal.

If this is a hardware signal, then ffh.fault_type has the value '04'b3. cfh.ret_pb or ffh.ret_pb points to the next instruction to be executed in the faulting procedure.

No information structure is available.

The default on-unit flushes the input and output buffers of the user's terminal, prints the message "QUIT." on the terminal, and calls a new command level.

▶ RECORD

(software, returnable)

This condition is raised when record size is different from the variable defined in the PL/I source. (Generally raised by full PL/I only. Not available through PL1G.)

▶ 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).

If the interrupted operation can be aborted, the subsystem's on-unit should perform a nonlocal GOTO back into the subsystem at the appropriate point.

If the QUIT$ occurred during an operation that must be completed, the on-unit should set the info.start_sw to '1'b, record the QUIT$ request within the subsystem and return. The REN command will then execute a START command which will restart the subsystem at the point of interrupt. When the operation is complete, the subsystem should then honor the recorded QUIT$ request.

The default on-unit returns without setting the info.start_sw. The REN command will then print a diagnostic and return since it assumes the stack held no subsystem able to accept reentry.

Information Structure:

```
DCL 1 info based
      2 start_sw bit(1) aligned;
```

▶ RESTRICTED_INST$

(hardware, returnable)

The process has attempted to execute an instruction whose use is restricted to ring-0 procedures. Certain of these instructions (in the I/O class) can be simulated by ring 0. An instruction which causes this condition to be raised could not be simulated by this mechanism.

ffh.fault_type	Value '00'b3.
ffh.ret-pb	Points to the faulting instruction.

▶ RO_ERR$

(software, returnable)

A ring-0 call to ERRPR$ or ERRRTN has been made, as the result of some fatal error condition having been detected.

No information structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

▶ SIZE

(software, not returnable)

This condition is raised when a program tries to do an arithmetic conversion and the value is too large to fit into the target data type. It can occur when converting either a floating-point number or a decimal integer to a binary integer.

The standard PL/I condition information structure is provided.

▶ STACK_OVF$

(hardware, returnable)

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.

ffh.fault_type Value '54'b3.

ffh.fault_addr The last stack segment in the chain of stack segments of the stack that overflowed. It is this segment that contains the zero extension pointer that caused the stack overflow fault.

ffh.ret_pb Points to the faulting instruction.

No information structure is available.

The static-mode default on-unit will attempt to simulate the Prime 300 fault handling for stack overflow fault if the appropriate word of segment '4000 is nonzero. (See the *System Architecture Reference Guide*.) If this word is zero or if no static-mode program is in execution, the standard default handling occurs.

▶ STOP$

(software, not returnable)

The process has executed a STOP statement in a higher-level-language program. This condition should not be handled by user programs, as it is used by Prime software to ensure the proper operation of the STOP statement in the various languages.

No information structure is available.

The default on-unit for this condition performs a nonlocal GOTO back to the command processor which invoked the procedure which (or one of the dynamic descendants of which) executed the STOP statement.

▶ STORAGE

(software, returnable)

This condition occurs when your program attempts to allocate storage and none is available. (It is generally raised by full PL/I only and is not available through PL1G.)

▶ STRINGRANGE

(software, returnable)

One argument of the SUBSTR function is out of range of the string.

▶ STRINGSIZE

(software, returnable)

The target of a string assignment is too small to contain the value. The default on-unit simply returns.

<u>Information Structure</u>:

The standard PL/I condition information structure is provided.

▶ SUBSCRIPTRANGE

A subscript is out of range.

<u>Information Structure</u>: Standard PL/I information structure.

▶ SVC_INST$

(hardware, returnable)

The process has executed an SVC instruction, but the system has not been able to perform the operation. If the user is in "SVC virtual" mode, all SVC instructions result in this condition being raised.

ffh.fault_type	Value '14'b3.
ffh.ret_pb	Points to the location following the SVC instruction.

Information Structure:

```
DCL 1 info based,
      2 reason fixed bin;
```

info.reason values

1 Bad SVC operation code or bad argument(s).

2 Alternate return needed but was 0.

3 Virtual SVC handling is in effect in this process.

For the case of virtual SVC's only (info.reason code of 3), the static-mode default on-unit will simulate the Prime 300 fault handling for the SVC fault, if the appropriate word of segment '4000 is nonzero. If this word is 0 or if there is no static-mode program in execution, the standard default handler prints a diagnostic and calls a new command level. (See the System Architecture Reference Guide for the exact location.)

▶ TRANSMIT

(software, returnable)

This condition occurs when data cannot be transmitted reliably between a data set and PL/I storage. (It is generally raised by full PL/I only and is not available through PL1G.)

▶ UII$

(hardware, returnable)

The process has executed an unrecognized instruction that nevertheless caused an unimplemented instruction fault, or else the system UII handler detected an error in processing the valid UII.

The fault frame header that accompanies this condition is nonstandard in that ffh.regs is not valid: the registers at time of fault are unavailable.

ffh.ret_pb	Points to the next instruction to be executed in the faulting procedure.

▶ UNDEFINEDFILE (file)

(software, not returnable)

This condition is raised when an OPEN statement cannot associate an input file with an existing PRIMOS file or device. The default on-unit prints a message and signals the ERROR condition.

▶ UNDEFINED_GATE$

(software, not returnable)

This condition is signalled when the process has called an inner ring gate segment at an address within the initialized portion of the gate segment, but there was no legal gate at that address. This error can arise because gate segments are padded, from the last valid gate entry to the next page boundary, with "illegal" gate entries.

No information structure is available.

▶ UNDERFLOW

(hardware, returnable)

This condition is signalled when the result of the floating-point binary or decimal calculation is too small for representation. The default on-unit sets the floating-point accumulator to 0.0e0. If the underflow occurred as a store exception, the affected portion of memory is also set to 0.0e0. The default on-unit returns and the calculation proceeds, using the 0.0e0 value.

The standard PL/I condition information structure is provided.

▶ ZERODIVIDE

(hardware, not returnable)

This condition is signalled when a division by 0 (floating-point or fixed-point) occurs. The default on-unit prints a message and signals the ERROR condition. For compatibility with earlier versions of PRIMOS, if the condition is the result of a floating-point operation, the user may type START following the printing of the message. The default on-unit will then set the register involved to the largest possible single-precision floating-point value and proceed with the calculation.

The standard PL/I condition information structure is provided.

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 options (short),
           2 ret_sb ptr options (short),
           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 GOTOs. 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 "nonsupport" 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

If '1'b, 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

If '1'b, 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 — If '1'b, 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

Programmers writing procedures 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 (EFH)

Any procedure (or begin block) which is 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 automatically by the FTN, F77, and PL1G compilers. PMA programs require explicit space allocation.

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 options (short),
           2 ret_sb ptr options (short),
           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 — If '1'b, 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 — If '1'b, 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 — If '1'b, this stack frame belongs to a "nonsupport" procedure; otherwise '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 — If '00'b, this frame was created by a regular procedure call; if '10'b, this frame is a fault frame (ffh) with valid saved registers; if '01'b, 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.seq_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 which 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.

on-unit_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 on-unit 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 not 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 seq_no bit(12),
      2 ret_pb ptr options (short),
      2 ret_sb ptr options (short),
      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,root.mbz 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 '10'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 '10'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
If '1'b, this on-unit was made via a call to the primitive MKONU$; if '0'b, it was made via the PL/I on statement.

flags.specify
If '1'b, 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
If '1'b, the snap option was specified in the PL/I on statement that created this on-unit; '0'b otherwise.

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.

PART VIII

Library Building and Management

23 Library Management

This chapter 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 such as FTN, COBOL, or PMA. These object-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 first been processed by LIBEDB.

LIBEDB is maintained as the runfile LIBEDB.SAVE 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 filename 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

Startup

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 displays 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.)

```
         { name, <SFL>, or <RFL> }
   COPY  { 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.

```
         { name, <SFL>, or <RFL> }
   FIND  { ALL                   }
```

Moves the pointer to the 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 and 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.)

Obsolete Commands

The following commands are outmoded but are included for the sake of compatibility:

ET

Writes an end-of-tape mark on the output file ('223, '223 on paper tape; 0 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 FILE1.BIN, FILE2.BIN, FILE3.BIN, and FILE4.BIN. Each file contains a single module, although FILE1.BIN and FILE2.BIN contain multiple entry points. The example shows the EDB commands to list the entry points of each file, plus the commands necessary to combine them into a library file, LIBEXP.

```
OK, EDB FILE1.BIN
[EDB REV 18.2]
ENTER, F ALL
ENT1A ENT1B ENT1C
.BOTTOM.
ENTER, NEWINF FILE2.BIN
ENTER, F ALL
ENT2D ENT2E
.BOTTOM.
ENTER, NEWINF FILE3.BIN
ENTER, F ALL
ENT3G
.BOTTOM.
ENTER, NEWINF FILE4.BIN
ENTER, F ALL
ENT4H
.BOTTOM.
ENTER, OPEN LIBEXP
ENTER, NEWINF FILE1.BIN
ENTER, RFL
ENTER, C ALL
ENT1A ENT1B ENT1C
.BOTTOM.
ENTER, I FILE2.BIN
ENTER, I FILE3.BIN
ENTER, I FILE4.BIN
ENTER, SFL
ENTER, QUIT
OK,
```

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. For example:

```
OK, EDB LIBEXP
[EDB REV 18.2]
ENTER, F ALL
<RFL> ENT1A ENT1B ENT1C ENT2D ENT2E ENT3G ENT4H <SFL>
.BOTTOM.
ENTER, TOP
ENTER, TERSE
FNTER, F ALL
 <RFL> ENT1A ENT2D ENT3G ENT4H <SFL>
.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 NFILE3.BIN containing entry points ENT3F and ENT3G. The output file is LIBNEW.

```
OK, EDB NFILE3.BIN
[EDB REV 18.2]
ENTER, F ALL
<RFL>  ENT3F ENT3G <SFL>
.BOTTOM.
ENTER, Q

OK, EDB LIBEXP LIBNEW
[EDB REV 18.2]
ENTER, R ENT3G NFILE3.BIN
<RFL> ENT1A ENT1B ENT1C ENT2D ENT2E ENT3G <SFL>
ENTER, C ALL
ENT4H
.BOTTOM.
ENTER, Q

OK, EDB LIBNEW
[EDB REV 18.2]
ENTER, F ALL
<RFL> ENT1A ENT1B ENT1C ENT2D ENT2E ENT3F ENT3G ENT4H <SFL>
.BOTTOM.
ENTER, Q
```

APPENDIXES

A

New File Management Subroutines for Rev. 19

NEW FEATURES IN REV. 19

ACLs (Access Control List System)

Several subroutines have been added at Rev. 19 to support Access Control Lists (ACLs):

Subroutine	Function
AC$CAT	Protect file system object with access category.
AC$CHG	Change contents of an ACL.
AC$DFT	Revert file system object to default protection.
AC$LIK	Copy ACL from one file system object to another.
AC$LST	List contents of an ACL.
AC$RVT	Convert an ACL directory to a password directory.
AC$SET	Create an ACL.
CALAC$	Calculate access on a file system object.
CAT$DL	Delete an access category.
CHG$PW	Change login validation password.

CREPW$	Create a new password directory.
DIR$LS	Search directories.
DIR$RD	Read directory entries sequentially.
ENT$RD	Read named directory entry.
FIL$DL	Delete a file.
GETID$	Return user's full ACL identity.
ISACL$	Determine type of a directory.
PA$DEL	Delete priority ACL.
PA$LST	List priority ACL.
PA$SET	Create priority ACL.
SGD$DL	Delete a segment directory entry.

Before using these subroutines, please read the section on access control in the Prime User's Guide for Rev. 19 or higher. Note also that the older subroutines RDEN$$ and SATR$$ have been modified for use with ACLs.

New Subroutines for Attaching

The following subroutines should be substituted for ATCH$$:

Subroutine	Function
AT$	Attach by pathname.
AT$ABS	Attach to top-level directory on specified partition.
AT$ANY	Attach to top-level directory on any partition.
AT$HOM	Set current directory as home directory.
AT$OR	Set home and/or current directory to origin.
AT$REL	Attach relative to current directory.

Date Retrieval

The following new subroutines retrieve or convert date and time:

Subroutine	Function
CV$DQS	Convert binary date to quadseconds.
CV$DFT	Convert formatted date to binary.
CV$FDA	Convert binary date to ISO format.
CV$FDV	Convert binary date to visual format.
DATE$	Return current date and time in binary format.

User Information

The following subroutines retrieve user information:

Subroutine	Function
USER$	Return process number and user count.
UTYPE$	Return type of current process.

DESCRIPTION OF THE SUBROUTINES

▶ AC$CAT

Purpose

Files may be added to an access category with the AC$CAT call.

Usage

```
DCL AC$CAT ENTRY (CHAR(128)VAR, CHAR(32)VAR, FIXED BIN);

CALL AC$CAT (object-path, category-name, code)
```

object-path	Pathname of the file system object to be protected (input).
category-name	Name of the category to which the object should be added (input).
code	Standard return code.

Discussion

The object must exist and must be a file, UFD, or segment directory. The category must exist in the same directory as the object and must be an access category. If the object is a password directory and its parent is an ACL directory, the object will be converted to an ACL directory.

Protect access is required on the parent directory, or on the object itself if it is a directory or access category. Use access is required at each intermediate name in the path. List access is also required on the parent. If the object is a password directory and protect access is not available on its parent, owner access is required on the object.

Before using this subroutine, please read the chapter on access control in the Prime User's Guide.

AC$CAT requires protect and list access to the parent of the file system object.

▶ AC$CHG

Purpose

Existing ACLs may be modified with the AC$CHG call.

Usage

```
DCL AC$CHG ENTRY (CHAR(128)VAR, PTR, FIXED BIN);

CALL AC$CHG (name, acl-ptr, code)
```

name	Pathname of the object whose ACL is to be modified (input).
acl-ptr	Pointer to the ACL structure (input). This structure is described with AC$LST.
code	Standard return code.

Discussion

AC$CHG is similar to AC$SET, but rather than replacing the entire contents of the old ACL, AC$CHG updates the existing ACL with the new data. The object to be changed must be an existing access category or a specifically protected file. (If it is not, an error is returned.) As in the ACL commands, if the access half of the access pair is null, the id is removed from the ACL. Otherwise, if the id already exists in the ACL its access list is simply changed, and if it does not exist it is added.

Before using this subroutine, please read the chapter on access control in the Prime User's Guide.

▶ AC$DFT

Purpose

A file may be given default protection with the AC$DFT call.

Usage

DCL AC$DFT ENTRY (CHAR(128)VAR, FIXED BIN);

CALL AC$DFT (name, code)

name	Name of the file system object whose protection is to change (input).
code	Standard return code.

Discussion

The object must exist and be a file, UFD, or segment directory. If it is a password directory and its parent is an ACL directory, it will be converted to an ACL directory. Attempts to use AC$DFT on MFDs will be rejected with error code E$IMFD (operation illegal on MFD).

AC$DFT requires protect and list access for the parent of the object, or on the object itself if it is a directory. Use rights are required at each intermediate node in the tree. List rights are also required on the parent. If the object is a password directory, owner access is required if protect access is not available on the parent.

▶ AC$LIK

Purpose

ACLs may be copied from one file to another with the AC$LIK routine. Thus one file may be given the same protection as another.

Usage

```
DCL AC$LIK ENTRY (CHAR(128)VAR, CHAR(128)VAR, FIXED BIN);

CALL AC$LIK (target-path, reference-path, code);
```

target-path	Pathname of file system object to be protected (input).
reference-path	Pathname of file system object from which to take ACL (input).
code	Standard return code.

Discussion

Both target and reference must be existing file system objects. A new specific ACL will be created with the ACL of the reference, regardless of how the target and reference are currently protected. If the target is a password directory and its parent is an ACL directory, the target will be converted to an ACL directory.

AC$LIK requires protect and list access to the target's parent, or protect access to the target. It also requires list access to the parent of the reference.

▶ AC$LST

Purpose

ACLs are read using AC$LST.

Usage

```
DCL AC$LST ENTRY (CHAR(128)VAR, PTR, FIXED BIN, CHAR(128)VAR, FIXED
                 BIN, FIXED BIN);

CALL AC$LST (name, acl-ptr, max-entries, acl-name, acl-type, code);
```

name — Pathname of the file system object for which information is desired (input).

acl-ptr — Pointer to return structure discussed below (input, points to output).

max-entries — Most entries that user's buffer can handle (input).

acl-name — Name of the ACL protecting the object (output). The name is determined by the algorithm listed under the Discussion.

acl-type — Type of the ACL protecting the object (output). Possible values are:

0	Specific ACL (spec_aclt)
1	Access category (cat_aclt)
2	Default access provided by specific ACL (dft_spec_aclt)

code — Standard return code.

Discussion

AC$LST requires list access to the parent of the file system object.

If the name is null, the contents of the default ACL for the current directory are returned. If max-entries is 0, only acl-name and acl-type are returned. The acl-name returned (which is a full pathname) is determined by the following algorithm:

```
acl_name(object) = If (object category_protected)
                   then category name
                   else if (object specific_protected)
                        then object name
                        else acl_name(parent(object))
```

acl-ptr points to a structure which looks like the following:

```
dcl 1 acl,
      2 version fixed bin,        /* Input, must be 2 */
      2 entry_count fixed bin,    /* Number of pairs */
      2 entries(entry_count)char(80) var;  /*<access_pair>s*/
```

▶ AC$RVT

Purpose

AC$RVT converts an ACL directory to a password directory.

Usage

```
DCL AC$RVT ENTRY (FIXED BIN);

CALL AC$RVT (code);
```

code — Standard error code (output). Possible values are:

E$NRIT	Protect access is not available.
E$NINF	List access is not available.
E$CATF	The directory contains one or more access categories.
E$ADRF	The directory contains one or more ACL subdirectories.
E$WTPR	The disk is write-protected.

Discussion

AC$RVT reverts the current directory to a password directory. The directory must not contain any access categories or ACL subdirectories; if it does the call will be rejected.

AC$RVT is provided for compatibility reasons only, and should be used sparingly, if at all.

Protect access is required on the current directory.

▶ AC$SET

Purpose

The AC$SET call provides user programs with a method of creating and replacing the ACL belonging to a category or file.

Usage

```
DCL AC$SET ENTRY (FIXED BIN, CHAR(128)VAR, PTR, FIXED BIN);

CALL AC$SET (key, name, acl-ptr, code);
```

key — Indicates caller's intentions (input). Possible values are:

- 0 — Create a new ACL if one does not exist; replace one if it does.
- K$CREA — Create a new ACL. If one already exists, return an error.
- K$REP — Replace the contents of an existing ACL. If one does not exist, return an error.

name — Pathname of the file system object to be protected (input).

acl-ptr — Pointer to the ACL structure (input). The acl-ptr points to a structure like that for AC$LST, above.

code — Standard return code.

Discussion

AC$SET requires protect and list access to the parent of the object, or protect access to the object itself.

The action taken by AC$SET is determined by the type of the object named in the call and by the key, as follows:

- The named object is an access category: if the key is K$CREA, an error is returned. Otherwise, the category's existing ACL is replaced with the new one pointed at by acl-ptr.
- The named object is a file: if the file is protected by a specific ACL and the key is K$CREA, an error is returned. Otherwise, a new specific ACL is created and the object is

pointed to it. Any old specific ACL is deleted. If the object is a password directory and its parent is an ACL directory, it will be converted to an ACL directory.

- The named object does not exist: if the key is not K$REP, a new access category is created with the given name and ACL. Otherwise, an error is returned.

▶ AT$

Purpose

AT$ does an attach by pathname.

Usage

```
DCL AT$ ENTRY (FIXED BIN, CHAR(128) VAR, FIXED BIN);

CALL AT$ (set-home-key, pathname, code);
```

set-home-key — A key indicating whether or not the home attach point should be set after the attach is completed (input). Possible values are:

K$SETH	Set home.
K$SETC	Do not set home.

pathname — Pathname of the directory which is to be attached to (input). If it is null, AT$ has the same effect as AT$HOM, below.

code — Standard error code (output). Possible values are:

E$BKEY	An illegal key value was passed.
E$ITRE	The treename was illegal.
E$FNTF	Some part of the pathname does not exist.
E$NRIT	Use rights were unavailable at some level.
E$NINF	Some node in the tree could not be accessed, and that node's parent was missing list access.

E$NATT	A relative attach was attempted, but the current attach point was invalid.

Discussion

AT$ provides the ability to do a pathname attach in one call. The pathname standard is followed:

- A leading "*" means attach relative to the home attach point.
- A partition name of "<*>" means current partition.
- A partition name of "<>" means any partition.
- A bare partition name indicates the MFD.

However, there are two exceptions:

- Backwards attaching (up the tree) is not supported.
- Pathnames beginning with an entryname are considered absolute.

Use access is required at each node in the tree, including the MFD.

If the directory is a password directory with both an owner and a nonowner password, and the supplied password matches neither, the BAD_PASSWORD$ condition is signalled, rather than an error code being returned. First there is a five-second delay to discourge machine-aided cracking of passwords.

▶ AT$ABS

Purpose

AT$ABS attaches to a top-level directory. It is used in place of ATCH$$ with the K$IMFD key and a positive logical device or disk number. AT$ABS uses partition names rather than LDEV numbers.

Usage

DCL AT$ABS ENTRY (FIXED BIN, CHAR(32)VAR, CHAR(39)VAR, FIXED BIN);

CALL AT$ABS (set-home-key, part-name, dir-name, code)

set-home-key	Indicates caller's intention (input). Possible values are the following.

K$SETH Set home as well as current (input).

K$SETC Set current directory only.

part-name Name of the disk partition on which the directory is to be found (input). The rules for names are given below.

dir-name Name of the directory, including the password, which should be separated from the directory name by a space (input).

code Standard return code.

Discussion

If the partition name is null, logical device 0 (the command device) is assumed. If the directory name is null, the MFD is assumed. If the name is "*", the current partition is searched.

▶ AT$ANY

Purpose

AT$ANY is used in place of ATCH$$ with the K$IMFD key and a logical device number of '100000. It attaches to a top-level directory on any partition.

Usage

```
DCL AT$ANY ENTRY (FIXED BIN, CHAR(39)VAR, FIXED BIN);

CALL AT$ANY (set_home_key, dir_name, code)
```

set_home_key If K$SETH, set home as well as current (input).

dir_name Name of the directory, including the password, which should be separated from the directory name by a space (input).

code Standard return code.

Discussion

All local partitions are searched first.

▶ AT$HOM

Purpose

AT$HOM sets the current directory to be the same as home.

Usage

DCL AT$HOM ENTRY (FIXED BIN);

CALL AT$HOM (code);

code	Standard return code.

Discussion

AT$HOM replaces an ATCH$$ call with a key of K$IMFD and a null name.

▶ AT$OR

Purpose

AT$OR sets the current UFD, and optionally the home UFD.

Usage

DCL AT$OR ENTRY (FIXED BIN, FIXED BIN);

CALL AT$OR (set-home-key, code);

set-home-key	If K$SETH, set home as well as current directory to initial attach point (input).
code	Standard return code.

▶ AT$REL

Purpose

AT$REL attaches relative to the current directory.

Usage

```
DCL AT$REL ENTRY (FIXED BIN, CHAR(39)VAR, FIXED BIN);

CALL AT$REL (set-home-key, dir-name, code);
```

set-home-key	If K$SETH, set home as well as current (input).
dir-name	Name of the directory, including the password, which should be separated from the directory name by a space (input).
code	Standard return code.

Discussion

AT$REL replaces ATCH$$ calls that used the K$ICUR key.

▶ CALAC$

Purpose

The CALAC$ function allows programs to determine the accesses available to the user on any given file system object.

Usage

```
DCL CALAC$ ENTRY (CHAR(128)VAR, PTR, CHAR(80)VAR,
                  CHAR(80)VAR, FIXED BIN) RETURNS(BIT(1));

have-access = CALAC$ (name, id-ptr, acc-needed, acc-gotten, code)
```

name	Pathname of the file system object to check (input).
id-ptr	Pointer to the user-id structure (input).
acc-needed	A list of accesses required (input).
acc-gotten	The list of accesses available (output).

code — Standard return code.

have-access — True if acc-needed is a subset of acc-gotten (returned).

Discussion

The user-id structure pointed to by id-ptr is the same as that for GETID$ below. If id-ptr is null (the usual case), the current user's id and groups are used.

The acc-needed and acc-gotten strings are in ASCII format. They are strings consisting of mnemonic access mode names or the special modes ALL and NONE.

If the name is null, the rights for the current directory are returned.

If the object is password-protected, password rights are returned. If the CALAC$ call is made on the current directory, the string "Owner" is returned if the user has owner rights, and "Non-owner" is returned if the user is attached with nonowner rights. For files, a string of the form "owner_rights> <non_owner_rights>" is returned, where the rights strings wil be either a combination of the characters R (read), W (write), and D (delete) or the special string NIL (no rights). For password-protected objects the acc-needed string is ignored and have-access is always set to true.

CALAC$ requires list access to the parent of the object.

▶ CAT$DL

Purpose

Access categories may be deleted with the CAT$DL call.

Usage

```
DCL CAT$DL ENTRY (CHAR(128)VAR, FIXED BIN);

CALL CAT$DL (name, code);
```

name — Name of the category to be deleted (input).

code — Standard return code.

Discussion

The name must exist and must specify an access category. Specific ACLs may not be explicitly deleted. They are deleted by the system when the file they protect either is deleted, is put into an access category, or reverts to default protection.

An access category that protects the MFD may not be deleted.

▶ CHG$PW

Purpose

CHG$PW changes the login validation password.

Usage

```
DCL CHG$PW ENTRY (CHAR(16)VAR, CHAR(16)VAR, FIXED BIN);

CALL CHG$PW (old-pw, new-pw, code);
```

old-pw — The user's current login validation password (input).

new-pw — The new password desired (input). Passwords may contain any characters except PRIMOS reserved characters. Lowercase alphabetic characters are mapped to uppercase by CHG$PW. At the System Administrator's option, null passwords may be disallowed.

code — Standard error code (output). Possible values are:

- E$BPAR — One of the passwords is illegal.
- E$BPAS — The old password passed does not match the actual password.
- E$WTPR — The disk is write-protected.

Discussion

CHG$PW allows a user to change the login validation password. This is the password that a user gives during the LOGIN command, and has nothing to do with directory passwords.

▶ CREPW$

Purpose

CREPW$ creates a new password directory.

Usage

```
DCL CREPW$ ENTRY (CHAR(32), FIXED BIN, CHAR(6), CHAR(6), FIXED BIN);

CALL CREPW$ (name, name-length, owner-pw, non-owner-pw, code);
```

name	Name of the directory to be created (input).
name-length	Length of the name in characters (input).
owner-pw	Password which must be used to attach with owner rights (input).
nonowner-pw	Password that must be used to attach with nonowner rights (input).
code	Standard error code (output). Possible values are:

E$BNAM	The supplied name is illegal.
E$BPAR	The name length is illegal.
E$EXST	An object with the given name already exists.
E$NRIT	Add rights were not available on the current directory.
E$WTPR	The disk is write-protected.
E$NINF	An error occurred, and list rights were not available on the current directory.
E$NATT	The current attach point is invalid.

Discussion

CREPW$ is used to create new directories. It always creates a password directory. Add access is required on the current directory.

▶ CV$DQS

Purpose

CV$DQS converts the binary date to quadseconds.

Usage

```
DCL CV$DQS ENTRY (FIXED BIN(31), FIXED BIN(31));

CALL CV$DQS (fs-date, quadseconds);
```

fs-date — The date to be converted (input). The format of a 32-bit encoded FS-format date is described below.

quadseconds — Date as expressed in quadseconds since midnight of January 1, 1901 (output). Quadseconds are groups of four seconds.

Discussion

CV$DQS is part of the PRIMOS standard date package. It takes a standard FS-format bit-encoded date and converts it to absolute quadseconds since midnight of January 1, 1901 (01-01-01.00:00:00).

FS-format dates are bit-encoded as defined by the following structure:

```
dcl 1 fs_date,
      2 year bit(7),
      2 month bit(4),
      2 day bit(5),
      2 quadseconds fixed bin(15);
```

year — Year modulo 100, with the exception that years 100-128 mean 2000-2028.

month — Month, from 1 for January to 12 for December.

day — Day of the month, from 1 to 31.

quadseconds — Number of quadseconds (groups of four seconds) elapsed since midnight of the date described by the above three fields.

If the date passed is invalid, -1 is returned in the quadseconds field.

▶ CV$DTB

Purpose

CV$DTB converts the formatted date to binary.

Usage

```
DCL CV$DTB ENTRY (CHAR(128)VAR, FIXED BIN(31), FIXED BIN);

CALL CV$DTB (ascii-date, fs-date, code);
```

ascii-date — The ASCII-formatted date to be converted (input). Legal formats are described below.

fs-date — The bit-encoded FS-format date returned. FS-format dates are described below.

code — Standard error code (output). Possible values are:

E$BPAR — The passed date string is illegal.

Discussion

CV$DTB is part of the PRIMOS standard date package. It converts an ASCII-formatted date to FS (bit-encoded) format.

Standard ASCII-format dates may have one of the following three formats:

YY-MM-DD.HH:MM:SS{.DOW}	(ISO format)
MM/DD/YY.HH:MM:SS{.DOW}	(USA format)
DD MMM YY HH:MM:SS{ Day-of-week}	(Visual format)

Omitted date fields are replaced by today's date information; omitted time fields are replaced by zeros. If the string is null, 0 is returned. The day-of-week field is checked for consistency only.

FS-format dates are bit-encoded as defined with CV$DQS.

▶ CV$FDA

Purpose

CV$FDA converts the binary date to ISO format.

Usage

```
DCL CV$FDA ENTRY (FIXED BIN(31), FIXED BIN, CHAR(21));

CALL CV$FDA (fs-date, day-of-week, formatted-date);
```

fs-date — Standard FS-format date as described below (input).

day-of-week — Ordinal day-of-week number (output). Sunday = 0, Monday = 1, etc.

formatted-date — ASCII-formatted date in ISO format, as described below (output).

Discussion

CV$FDA is part of the PRIMOS standard date package. It converts an FS-format date string to ISO format. The date returned is of the format "YY-MM-DD.HH:MM:SS.DOW".

ISO-format dates are designed primarily for machine readability. Dates which are to be read primarily by people should be converted with CV$FDV, below.

If the passed date is illegal, formatted-date will be set to "** invalid date **" and day-of-week will be -1.

FS-format dates are bit-encoded as defined with CV$DQS.

▶ CV$FDV

Purpose

CV$FDV converts the binary date to visual format.

Usage

```
DCL CV$FDV ENTRY (FIXED BIN(31), FIXED BIN, CHAR(28)VAR);

CALL CV$FDV (fs-date, day-of-week, formatted-date);
```

date — Standard FS-format date as described below (input).

day-of-week — Ordinal day-of-week number (output). Sunday = 0, Monday = 1, etc.

formatted-date — ASCII-formatted date in visual format, as described below (output).

Discussion

CV$FDV is part of the PRIMOS standard date package. It converts an FS-format date string to "visual" format. Visual-format dates are described below.

Visual-format dates are designed primarily to be read by users. Because they contain blanks and are not ordered in a strictly decreasing way, they are not particularly suited for machine readability. Dates which are to be mainly machine-read should be converted with CV$FDA, above.

The date returned is of the format "DD MMM YY HH:MM:SS day-of-week".

If the passed date is illegal, formatted-date will be set to "** invalid date **" and day-of-week will be -1.

▶ DATE$

Purpose

DATE$ returns the current date and time in binary format.

Usage

```
DCL DATE$ ENTRY RETURNS (FIXED BIN(31));

fs-date = DATE$();
```

fs-date   Standard FS-format date as described below (output).

Discussion

DATE$ is part of the PRIMOS standard date package. It returns the current date and time in the standard bit-encoded FS format described below.

FS-format dates are bit-encoded as defined with CV$DQS.

▶ DIR$LS

Purpose

DIR$LS is a general-purpose directory searcher.

Usage

```
DCL DIR$LS ENTRY (FIXED BIN, FIXED BIN, BIT(1), BIT(4), PTR,
                  FIXED BIN, PTR, FIXED BIN, FIXED BIN,
                  FIXED BIN, (4) FIXED BIN, FIXED BIN(31),
                  FIXED BIN(31), FIXED BIN);

CALL DIR$LS  (dir-unit, dir-type, initialize, desired-types,
             wild-ptr, wild-count, return-ptr, max-entries,
             entry-size, ent-returned, type-counts,
             before-date, after-date, code);
```

dir-unit   Unit on which the directory to be searched is open (input).

dir-type — Type of object open on dir-unit. Legal values are:

2	SAM segment directory.
3	DAM segment directory.
4	Directory.

initialize — If set, the directory is to be reset to the beginning; otherwise, it is searched from the current position. This is useful so that large directories may be dealt with in more than one call, thus making a huge buffer area in the caller's routine unnecessary.

desired-types — A bit-encoded field defining what types of directory entries the caller wishes to have returned (input). In the following table, if the bit is set the specified type will be returned:

'1000'b	Directories.
'0100'b	Segment directories.
'0010'b	Files.
'0001'b	Access categories.

If all bits are set, type is not used as a selection criterion.

wild-ptr — Pointer to list of wildcard names for which to search (input). The list should be an array of char(32) varying strings. Wildcards are explained in the Prime User's Guide.

wild-count — Number of names in list pointed to by wild-ptr (input). If wild-count is 0, entryname is not used as a selection criterion.

return-ptr — Pointer to caller's return structure. The data structure returned by DIR$LS is described below. (Input, points to output.)

max-entries — Maximum number of entries that caller's structure can handle (input).

entry-size — Number of words reserved for each directory entry in caller's structure. max-entries * entry-size defines the size of the caller's structure in words (input). In Rev. 19, the normal size of a directory entry as returned by DIR$LS is 24 words.

entr-returned Number of entries returned by DIR$LS (output). This number will always be less than or equal to max-entries.

type-counts Number of entries of each type returned by DIR$LS. Counts are returned in the order files, segment directories, directories, access categories.

before-date Entries with date/time modified earlier than this date are selected by DIR$LS (input). The date should be in standard FS format, described with CV$DQS.

If the value of before-date is 0, it is not used as a selection criterion.

after-date Entries with date/time modified later than this date are selected by DIR$LS (input). The date should be in standard FS format, described with CV$DQS.

If the value of after-date is 0, it is not used as a selection criterion.

code Standard error code (output). Possible values are:

E$BUNT	dir-unit specified an illegal unit number.
E$UNOP	dir-unit is not open.
E$EOF	There are no more entries in the directory.

Discussion

DIR$LS is a general-purpose directory scanner. It selects directory entries by name (handling wildcards), type, and date/time modified (DTM). It may be used to search segment directories.

The directory must have been previously opened on some unit with one of the standard PRIMOS file-opening routines. List access is required to open directories.

The directory is searched sequentially from its beginning (if the initialize bit was set) or from the current position (if it was not). As each entry is read, it is checked against all of the selection criteria. If the entry meets all the criteria, it is copied into the caller's buffer. The search ends when there are no more entries in the directory or the caller's buffer becomes full, whichever occurs first.

All entries in the directory are returned if wild-count, before-date and after-date are 0, and desired-types is '1111'b.

The structure of a returned directory entry is:

```
dcl 1 dir_entry,
  2 ecw,
   3 type bit(8),
   3 length bit(8),
  2 entryname char(32) var,
  2 protection,
   3 owner_rights,
    4 spare bit(5),
     4 delete bit(1),
     4 write bit(1),
     4 read bit(1),
    3 delete_protect bit(1),
    3 non_owner_rights,
     4 spare bit(4),
     4 delete bit(1),
     4 write bit(1),
     4 read bit(1),
  2 file_info,
   3 long_rat_hdr bit(1),
   3 dumped bit(1),
   3 dos_mod bit(1),
   3 special bit(1),
   3 rwlock bit(2),
   3 spare bit(2),
   3 type bit(8),
  2 dtm like fs_date,
  2 non_default_acl bit(1) aligned,
  2 spare bit(16) aligned;
```

ecw.type — Entry Control Word for the entry:

- 2 — Normal directory entry (file, directory, or segment directory).
- 3 — An access category.

length — This field will always have a value of 24 in rev. 19.

name — Name of the entry.

owner_rights — The rights granted to users when attached to the containing directory with owner rights.

delete_protect — The setting of the ACL delete-protect switch. If this bit is on, the file may not be deleted. The bit may be reset by a call to the SATR$$ subroutine.

non_owner_rights The rights granted to users when attached to the containing directory with nonowner rights.

long_rat.hdr If set, indicates that the file is a Disk Record Availability Table (DSKRAT) containing more than one record.

dumped If set, the file has been backed up by MAGSAV.

dos_mod If set, the file was modified while PRIMOS II (DOS) was running.

special If set, the file is special (e.g. DSKRAT, BOOT, MFD) and may not be deleted.

rwlock Indicates the setting of the file's read/write concurrency lock. Values are:

0 Use system default setting.

1 Unlimited readers or one writer (excl).

2 Unlimited readers and one writer (updt).

3 Unlimited readers and writers (none).

file_info.type Indicates the type of object described by this entry. Possible values are:

0 SAM file.

1 DAM file.

2 SAM segment directory.

3 DAM segment directory.

4 Directory.

6 Access category.

dtm The date/time the file was last modified, in standard FS format. FS-format dates are described with CV$DQS.

non_default_acl This bit is set if the object is not protected by the default ACL; that is, it is protected by a specific ACL or by an access category.

▶ DIR$RD

Purpose

DIR$RD reads the contents of a directory sequentially, entry by entry.

Usage

```
DCL DIR$RD ENTRY (FIXED BIN, FIXED BIN, PTR, FIXED BIN,
                  FIXED BIN);
```

```
CALL DIR$RD (key, unit, return-ptr, max-return-len, code);
```

key Indicates what to do (input):

K$INIT Initialize to directory header.

K$READ Read from current position.

unit Unit number on which directory is open; list access must be available on the directory (input).

return-ptr Pointer to user's buffer (input, points to output).

max-return-len Size of user's buffer (input).

code Standard return code.

Discussion

The return-ptr points to a structure with the following format. See RDEN$$ for a non-PL1G description of the structure.

```
dcl 1 dir_entry based,
      2 ecw,
        3 type bit(8),
        3 len bit(8),
      2 name char(32),
      2 pw_protection bit(16) aligned,
      2 non_dft_prot bit(1) aligned,
      2 file_info,
        3 long_rat_hdr bit(1),
        3 dumped bit(1),
        3 dos_mod bit(1),
        3 special bit(1),
        3 rwlock bit(2),
        3 spare bit(2),
        3 type bit(8),
        2 dtm,
          3 date,
            4 year bit(7),
            4 month bit(4),
            4 day bit(5),
          3 time fixed bin,
        2 spare(2) fixed bin;
```

All entries are as defined in the description of the subroutine RDEN$$ except for non_dft_prot, which is set to true if the entry is not default-protected (that is, is protected specifically or by a category).

DIR$RD only returns entries for named objects. Thus, unlike RDEN$$, it will not return the ecw (Entry Control Word) for the directory header. The types are 2 for a file or directory, and 3 for an access category.

Note

Calls to DIR$RD and ENT$RD should not be made on the same directory file unit unless DIR$RD is called with the K$INIT key following each ENT$RD call.

▶ ENT$RD

Purpose

FNT$RD returns the contents of a directory entry specified by name.

Usage

```
DCL ENT$RD ENTRY (FIXED BIN, CHAR(32)VAR, PTR, FIXED BIN,
                  FIXED BIN);
```

```
CALL ENT$RD (unit, name, return-ptr, max-return-len, code);
```

unit	Unit number on which the directory is open (list access is required; input).
name	Name of the entry to read (input).
return-ptr	Pointer to return structure (input, points to output).
max-return-len	Size of user's buffer (input).
code	Standard return code.

Discussion

ENT$RD is identical to DIR$RD in what it returns, but rather than going sequentially through the directory, ENT$RD returns data for a particular named entry.

The structure returned by ENT$RD is identical to that returned by DIR$RD. As noted above, however, ENT$RD and DIR$RD should not be used together on the same file unit.

▶ FIL$DL

Purpose

FIL$DL deletes a file.

Usage

DCL FIL$DL ENTRY (CHAR(128)VAR, FIXED BIN);

CALL FIL$DL (object-name, code);

object-name Pathname of the object to be deleted (input).

code Standard error code(output). Possible values are:

E$ITRE object-name is not a legal treename.

E$NRIT Delete access was not available on the parent, or use access was missing from some intermediate node.

E$WTPR The disk is write-protected.

E$NINF An error occurred when searching for the file, and the directory level at which the error occurred did not allow list access.

E$DLPR The file's delete-protect switch is set.

Discussion

FIL$DL is used to delete files and empty directories. Delete access is required on the parent directory.

If error code E$DLPR is returned, SATR$$ must be called to reset the delete-protect switch before the file may be deleted. This error code will only be returned if the caller has delete access on the parent directory and may thus reset the delete-protect switch.

▶ GETID$

Purpose

The GETID$ call returns the user's id and groups.

Usage

```
DCL GETID$ ENTRY (PTR, FIXED BIN, FIXED BIN);

CALL GETID$ (id-ptr, max-groups, code);
```

id-ptr — Pointer to the full_id structure below (input, points to output).

max-groups — Maximum number of groups that the caller's full_id structure can handle (input).

code — Standard return code.

Discussion

The structure pointed to by id-ptr looks like:

```
    dcl 1 full_id
           2 version fixed bin,
           2 user_id char(32) var,
           2 group_count fixed bin,
           2 groups(group_count) char(32) var;
```

version — Version number of the structure. This must be supplied by the caller and must be 2 in Rev. 19.

user_id — The id of the current user.

group_count — Number of groups returned to the caller. This will always be the minimum of max-groups as supplied by the user and the number of groups the user has. In Rev. 19, users may have up to 32 groups. If max-groups is 0, this field is not returned.

groups — The list of groups currently valid for the user.

▶ ISACL$

Purpose

This is a function call. For purposes of compatibility ACL directories and password directories have the same type (as returned to users; internally they are different). Therefore, some method of distinguishing between the two is needed. ISACL$ returns PL1G true if the directory specified is an ACL directory.

Usage

```
DCL ISACL$ ENTRY (FIXED BIN, FIXED BIN) RETURNS (BIT(1));

is-acl-dir = ISACL$ (unit, code);
```

unit — File unit to check (input). unit is either a file unit number, or one of the following:

-1	Current directory
-2	Home directory
-3	Initial directory

code — Standard return code (output).

is-acl-dir — True if directory on unit is an ACL directory (returned).

Discussion

Before using this subroutine, please read the section on access control in the Prime User's Guide.

▶ PA$DEL

Purpose

Priority ACLs are removed with the PA$DEL CALL, callable only by user 1 and the System Administrator.

Usage

```
DCL PA$DEL ENTRY (CHAR(32)VAR, FIXED BIN);

CALL PA$DEL (partition-name, code);
```

partition-name Name of the partition from which to remove a priority ACL (input).

code Standard return code (output).

Discussion

Before using this subroutine, please read the section on access control in the Prime User's Guide.

▶ PA$LST

Purpose

Priority ACLs may be read by any user with the PA$LST call.

Usage

```
DCL PA$LST ENTRY (CHAR(128)VAR, PTR, FIXED BIN, FIXED BIN);

CALL PA$LST (name, acl-ptr, max-entries, code)
```

name Name of any object on the partition whose priority ACL is to be read (input).

acl-ptr Points to return structure described with AC$LST (input, points to output).

max-entries Most entries caller can handle (input).

code Standard return code (output).

Discussion

Normally, some access to the partition is required in order to determine the logical device number and through it get the priority ACL. Since it is possible to disallow all access to a partition with priority ACLs, however, PA$LST may be called with only a partition name (in angle brackets). In that case, it will merely look the partition up in the disk table and no access is required.

▶ PA$SET

Purpose

Priority ACLs may be added to a partition with the PA$SET call, which may be used only by user 1 and the System Administrator.

Usage

```
DCL PA$SET ENTRY (CHAR(32)VAR, PTR, FIXED BIN);

CALL PA$SET (partition-name, acl-ptr, code);
```

partition-name Name of the partition to be protected (input).

acl-ptr Pointer to ACL structure (input).

code Standard return code (output).

Discussion

The acl-ptr points to an ACL structure as for AC$LST. Any existing priority ACL on the specified partition will be replaced by the new one. If no REST$ entryb is in the ACl passed to PA$SET, no REST:NONE will be supplied.

Before using this call, please read the section on access control in the Prime User's Guide.

▶ SGD$DL

purpose

SGD$DL deletes a segment directory entry.

Usage

```
DCL SGD$DL ENTRY (FIXED BIN, FIXED BIN);

CALL SGD$DL (segdir-unit, code);
```

segdir-unit Unit on which the segment directory is open (input).

code Standard error code (output). Possible values are:

E$BUNT	segdir-unit contained an illegal value.
E$SUNO	The unit was not open, or was not open for writing.
E$NTSD	The object open on segdir-unit was not a segment directory.
E$FNTS	The entry at the current position did not exist, or the segment directory was positioned past the end.

Discussion

SGD$DL is used to delete an entry from a segment directory. The segment directory must have been previously opened for writing (by a module such as SRCH$$), and must be positioned at the entry to be deleted (by SGDR$$).

▶ USER$

Purpose

USER$ returns process number and user count.

Usage

```
DCL USER$ ENTRY (FIXED BIN, FIXED BIN);

CALL USER$ (current-user-number, user-count);
```

current-user-number User number of the process issuing the call (output).

user-count Total number of users logged into the system (output).

Discussion

USER$ returns the user number of the current process, and the total number of users logged into the system.

▶ UTYPE$

Purpose

UTYPE$ returns the type of the current process.

Usage

```
DCL UTYPE$ ENTRY (FIXED BIN);

CALL UTYPE$ (user-type)
```

user-type   Type of the process making the call (output). User types are defined below.

Discussion

UTYPE$ returns the user type of the current process. The user type identifies the process by certain classes defined below. It is the preferred method of determining whether or not a given process is a phantom.

The possible user types are:

U$NORM	Local terminal user.
U$TREM	User gone to a remote system.
U$FREM	User from a remote system.
U$THRU	User logged through (both to and from remote).
U$SUSR	Supervisor (user 1).
U$TFAM	FAM I running at a user terminal.
U$PH	Cominput-style phantom.
U$CPH	CPL-style phantom.
U$NPX	NPX slave.
U$PFAM	FAM I running as a phantom.
U$NET	Network server process (NETMAN).

There are also four special types that mark the ranges of terminal and nonterminal (phantom) users. These markers are:

U$LTUT Lowest terminal user type.

U$HTUT Highest terminal user type.

U$LPUT Lowest phantom user type.

U$HPUT Highest phantom user type.

By using these marker types, callers can avoid having to change the range they check when new types are added to the list.

B
Message Facility Subroutines

See appendix P

INTRODUCTION

The Primos MESSAGE command has been extended to include calls for sending and receiving interuser messages. The subroutines may also set and query a user's willingness to receive messages. Messages may be sent in either immediate or deferred mode (to be delivered at command level only), and may be addressed with either a user name or a user number. Reception may also be controlled, allowing users to select one of three modes of reception: receive at any time, receive at command level only, or never receive.

The subroutines that support the message facility are:

Subroutine	Function
SMSG$	Send an interuser message.
RMSGD$	Receive a deferred message.
MGSET$	Set the receiving state for messages.
MSG$ST	Return the receiving state of a user.

▶ SMSG$

Purpose

SMSG$ sends an interuser message.

Usage

```
CALL SMSG$(key,name,namlen,number,reserv,rsvlen,text,txlen,ervec)
```

All parameters are INTEGER*2.

key — User option:

- 0 Deferred message.
- 1 Immediate message.

name — User name of addressee. It is blank if message is addressed by user number or if message is to the operator.

namlen — Length of name in characters.

number — User number of addressee. It is 0 if message is addressed by user name or if message is to the operator.

resrv — Reserved, must be 0.

rsvlen — Reserved, must be 0.

text — Text of message to be sent, may contain a terminating NL (octal 212).

txlen — Length of text in characters, between 1 and 79.

ervec — Returned error code:

ervec(1) Return code:

- E$NRCV Requires that receive be enabled.
- E$UADR Unknown addressee.
- E$UDEF User unable to receive messages.
- E$PRTL Operation was partially blocked.

E$NSUC Operation unsuccessful.

0 Operation successful.

ervec(2) Number of users configured on the system or length of the portion of ervec(4)-(n).

ervec(3) Status of link:

XS$CLR Connect cleared.

XS$BPM Unknown node address.

XS$DWN Node not responding.

ervec(4-131) User status:

E$UBSY User busy, please wait.

E$UNRV User not receiving now. The position in the vector minus three is the number of the user causing the returned code.

Note that this portion of ervec is optional depending on the value of ervec(2) supplied.

Discussion

Messages may be addressed with either a user name or a user number. If both are supplied, the user number will be used. If a only a user name is supplied, all users with the specified user name will receive the message. If user number is supplied, the process with that user number will receive the message.

Additionally, messages may not be sent to phantoms by their user names. Deferred messages sent to the user number of a phantom will go into the COMOUTPUT file of that phantom.

▶ RMSGD$

Purpose

RMSGD$ receives a deferred message.

Usage

```
CALL RMSGD$(sender,sndlen,sndnum,reserv,rsvlen,time,text,txtlen)
```

All parameters are INTEGER*2.

sender	User name of sender.
sndlen	Length of sender buffer in characters.
sndnum	User number of sender.
reserv	Reserved, must be 0.
rsvlen	Reserved, must be 0.
time	Time message was sent (minutes past midnight).
text	Text of message.
txtlen	Length of text buffer in characters.

▶ MGSET$

Purpose

MGSET$ sets the receiving state for messages.

Usage

```
CALL MGSET$(key,code)
```

Both parameters are INTEGER*2.

key User option:

K$ACPT	Accept all messages.
K$DEFR	Accept deferred messages only.

	K$RJCT	Reject all messages.
code	Return code:	
	E$BKEY	Bad key.
	0	No error.

▶ MSG$ST

Purpose

MSG$ST returns the receiving state of a user.

Usage

```
CALL MSG$ST(key,number,reserv,rsvlen,name,namlen,state)
```

All parameters are INTEGER*2.

key	K$READ = return user's name and state.	
number	User number of process for which state is desired.	
reserv	Reserved, must be 0.	
rsvlen	Reserved, must be 0.	
name	User name of process.	
namlen	Length of name buffer supplied (characters).	
state	Returned status:	
	K$ACPT	Accepting all messages.
	K$DEFR	Accepting deferred messages only.
	K$RJCT	Rejecting all messages.
	K$NONE	User does not exist.
	K$BKEY	Invalid state because key is bad.
	K$BREM	Invalid state because reserved field is bad.

C
Keys (SYSCOM>KEYS.INS)

INTRODUCTION

This Appendix summarizes the keys associated with PRIMOS subroutine calls. Use of these keys is explained in Chapter 2, and in the chapter for each calling language.

All key values here are given in decimal notation, while the SYSCOM file listing uses some octal notation.

```
C KEYS.INS.FTN, PRIMOS>INSERT, PRIMOS GROUP, 01/04/82
      /*************************************************************/
      /*                                                           */
      /*       KEY DEFINITIONS                                     */
      /*                                                           */
      /******************** PRWF$$ ********************            */
      /*              ****** RWKEY  ******                         */
         K$READ =  1,      /* READ                                 */
         K$WRIT =  2,      /* WRITE                                */
         K$POSN =  3,      /* POSITION ONLY                        */
         K$TRNC =  4,      /* TRUNCATE                             */
         K$RPOS =  5,      /* READ CURRENT POSITION                */
      /*              ****** POSKEY ******                         */
         K$PRER =  0,      /* PRE-POSITION RELATIVE                */
         K$PREA =  8,      /* PRE-POSITION ABSOLUTE                */
         K$POSR =  16,      /* POST-POSITION RELATIVE              */
         K$POSA =  24,      /* POST-POSITION ABSOLUTE              */
      /*              ****** MODE   ******                         */
         K$CONV =  256,   /* CONVENIENT NUMBER OF WORDS            */
         K$FRCW =  16384, /* FORCED WRITE TO DISK                  */
```

```
/*                                                            */
/********************* SRCH$$ *********************           */
/*                ****** ACTION ******                        */
/* K$READ =  1,     /* OPEN FOR READ                          */
/* K$WRIT =  2,     /* OPEN FOR WRITE                         */
   K$RDWR =  3,     /* OPEN FOR READING AND WRITING           */
   K$CLOS =  4,     /* CLOSE FILE UNIT                        */
   K$DELE =  5,     /* DELETE FILE                            */
   K$EXST =  6,     /* CHECK FILE'S EXISTENCE                 */
   K$VMR  =  16,    /* OPEN FOR VMFA READING                  */
   K$VMRW =  48,    /* OPEN FOR VMFA READING/WRITING          */
   K$GETU =  16384, /* SYSTEM RETURNS UNIT NUMBER             */
/*                ****** REF    ******                        */
   K$IUFD =  0,     /* FILE ENTRY IS IN UFD                   */
   K$ISEG =  64,    /* FILE ENTRY IS IN SEGMENT DIRECTORY     */
   K$CACC =  512,  /* CHANGE ACCESS                           */
/*                ****** NEWFIL ******                        */
   K$NSAM =  0,     /* NEW SAM FILE                           */
   K$NDAM =  1024,  /* NEW DAM FILE                           */
   K$NSGS =  2048,  /* NEW SAM SEGMENT DIRECTORY              */
   K$NSGD =  3072,  /* NEW DAM SEGMENT DIRECTORY              */
   K$CURR =  -1,   /* CURRENTLY ATTACHED UFD                  */
/*                                                            */
/*                                                            */
/********************* VINIT$ *********************           */
/*                                                            */
   K$ANY  =  0,
   K$CNSC =  8,        /* CONSECUTIVE SEGMENTS REQUIRED       */
   K$GATE =  1,        /* GATE ACCESS ON SEGMENT              */
   K$R    =  2,        /* READ ACCESS ON SEGMENT (^= K$READ!)*/
   K$RW   =  3,        /* READ/WRITE ACCESS ON SEGMENT        */
   K$RX   =  6,        /* READ/EXECUTE ACCESS                 */
   K$RWX  =  7,        /* READ/WRITE/EXECUTE                  */
/*                                                            */
/********************* GETSN$ *********************           */
/*                                                            */
   K$DOWN =  0,        /* ALLOCATE DECREASING SEGMENT #'S     */
   K$UP   =  1,        /* ALLOCATE INCREASING SEGMENT #'S     */
   K$UPC  =  2,        /* ALLOCATE INCREASING CONSEC. SEGS    */
   K$DWNC =  4,        /* ALLOCATE DECREASING CONSEC. SEGS    */
/*                                                            */
/********************* ATCH$$ *********************           */
/*                ****** KEY    ******                        */
   K$IMFD =  0,     /* UFD IS IN MFD                          */
   K$ICUR =  2,     /* UFD IS IN CURRENT UFD                  */
/*                ****** KEYMOD ******                        */
   K$SETC =  0,     /* SET CURRENT UFD (DO NOT SET HOME)      */
   K$SETH =  1,     /* SET HOME UFD (AS WELL AS CURRENT)      */
/*                ****** NAME   ******                        */
   K$HOME =  0,     /* RETURN TO HOME UFD (KEY=K$IMFD)        */
/*                ****** LDISK  ******                        */
   K$ALLD =  0,    /* SEARCH ALL DISKS                        */
/* K$CURR =  -1    /* SEARCH MFD OF CURRENT DISK              */
/*                                                            */
/* *********************** AC$SET **********************        */
/*                                                              */
/* K$ANY  =  0,      /* Do it regardless                        */      19
   K$CREA =  1,  /* Create new ACL (error if already exists)    */
   K$REP  =  2,      /* Replace existing ACL                    */
/*                      (error if does not exist)               */
/*                                                              */
/********************** SGDR$$ ********************             */
/*                   ****** KEY     ******                      */
   K$SPOS =  1,      /* POSITION TO ENTRY NUMBER IN SEGDIR      */
   K$GOND =  2,      /* POSITION TO END OF SEGDIR               */
   K$GPOS =  3,      /* RETURN CURRENT ENTRY NUMBER             */
   K$MSIZ =  4,      /* MAKE SEGDIR GIVEN NR OF ENTRIES         */
   K$MVNT =  5,      /* MOVE FILE ENTRY TO DIFFERENT POSITION   */
   K$FULL =  6,      /* POSITION TO NEXT NON-EMPTY ENTRY        */
   K$FREE =  7,      /* POSITION TO NEXT FREE ENTRY             */
/*                                                              */
/********************** RDEN$$ ********************             */
/*                   ****** KEY     ******                      */
/* K$READ =  1,      /* READ NEXT ENTRY                         */
   K$RSUB =  2,      /* READ NEXT SUB-ENTRY                     */
/* K$GPOS =  3,      /* RETURN CURRENT POSITION IN UFD          */
   K$UPOS =  4,      /* POSITION IN UFD                         */
   K$NAME =  5,      /* READ ENTRY SPECIFIED BY NAME            */
/*                                                              */
/************************* DIR$RD ************************       */
/*                                                              */
/* K$READ =  1,      /* Read next entry                         */
   K$INIT =  2,      /* Initialize directory (read header)      */
/*                                                              */
/********************** SATR$$ ********************             */
/*                   ****** KEY     ******                      */
   K$PROT =  1,      /* SET PROTECTION                          */
   K$DTIM =  2,      /* SET DATE/TIME MODIFIED                  */
   K$DMPB =  3,      /* SET DUMPED BIT                          */
   K$RWLK =  4,      /* SET PER FILE READ/WRITE LOCK            */
   K$SOWN =  5,      /* SET OWNER FIELD ON FILE                 */
   K$SDL  =  6,      /* SET ACL/DELETE SWITCH ON FILE           */
/*                   ****** RWLOCK ******                       */
   K$DFLT =  0,      /* Use system default value                */
   K$EXCL =  1,      /* N readers OR one writer                 */
   K$UPDT =  2,      /* N readers AND one writer                */
   K$NONE =  3,      /* N readers AND N writers                 */
/*                                                              */
/********************** ERRPR$ ********************             */
/*                   ****** KEY     ******                      */
   K$NRTN =  0,      /* NEVER RETURN TO USER                    */
   K$SRTN =  1,      /* RETURN AFTER START COMMAND              */
   K$IRTN =  2,      /* IMMEDIATE RETURN TO USER                */
/*                                                              */
/********************** GPATH$ *******************************/
/*                   ****** KEY     ******                      */
   K$UNIT =  1,      /* PATHNAME OF UNIT RETURNED               */
   K$CURA =  2,      /* PATHNAME OF CURRENT ATTACH POINT        */
```

```
       K$HOMA =  3,     /* PATHNAME OF HOME ATTACH POINT         */
       K$INIA =  4,     /* Pathname of initial attach point      */
    /*                                                           */
    /********************* MSG$ST ********************************/
    /*                                                           */
       K$ACPT = 0,      /* ACCEPT MSGS (ALSO MGSET)              */
       K$DEFR = 1,      /* DEFER MSGS  (ALSO MGSET)              */
       K$RJCT = 2,      /* REJECT MSGS (ALSO MGSET)              */
    /*                                                           */
    /********************** FNSID$ *******************************/
    /*                                                           */
       K$LIST = 1,      /* Return entire list                    */
       K$ADD  = 2,      /* Add to existing list                  */
       K$SRCH = 3,      /* Search for specific node              */
19  /*                                                           */
    /*************** FNCHK$, TNCHK$, IDCHK$, PWCHK$ **************/
    /*                                                           */
       K$UPRC = 1,      /* Mask string to uppercase              */
       K$WLDC = 2,      /* Allow wildcards (not PWCHK$)          */
       K$NULL = 4,      /* Allow null names                      */
       K$NUM  = 8,      /* Allow numeric names (FNCHK$ only)     */
       K$GRP  = 8,      /* Check group name (IDCHK$ only)        */
    /*                                                           */
    /*************************** Q$SET ***************************/
    /*                                                           */
       K$SMAX = 1       /* Set max quota                         */
    /*************************************************************/
    LIST
```

~~D~~

Error Handling

See appendix O

INTRODUCTION

This appendix defines PRIMOS error messages and codes, and error-handling conventions for Rev. 17 and later.

ERROR CODES

In most languages, error codes may be treated as data names rather than as numbers. See the chapter on your language for a discussion. The following table defines the error code names available for FORTRAN 77, FORTRAN IV, PMA, Pascal, and PL1G.

```
/* ERRD.INS.PLP, PRIMOS>INSERT, PRIMOS GROUP, 12/14/81
   MNEMONIC CODES FOR FILE SYSTEM (PL1)
   Copyright (c) 1981, Prime Computer, Inc., Natick, MA 01760 */
*******************************************************************/


       /*************************************************************/
       /*                                                          */
       /*                                                          */
       /*      ERROR CODE DEFINITIONS                              */
       /*                                                          */
       /   *                                                        */
       E$EOF  BY  00001, /* END OF FILE                  PE        */
       E$BOF  BY  00002, /* BEGINNING OF FILE            PG        */
       E$UNOP BY  00003, /* UNIT NOT OPEN                PD,SD     */
       E$UIUS BY  00004, /* UNIT IN USE                  SI        */
       E$FIUS BY  00005, /* FILE IN USE                  SI        */
       E$BPAR BY  00006, /* BAD PARAMETER                SA        */
       E$NATT BY  00007, /* NO UFD ATTACHED              SL,AL     */
       E$FDFL BY  00008, /* UFD FULL                     SK        */
       E$DKFL BY  00009, /* DISK FULL                    DJ        */
       E$NRIT BY  00010, /* NO RIGHT                     SX        */
       E$FDEL BY  00011, /* FILE OPEN ON DELETE          SD        */
       E$NTUD BY  00012, /* NOT A UFD                    AR        */
       E$NTSD BY  00013, /* NOT A SEGDIR                 —         */
       E$DIRE BY  00014, /* IS A DIRECTORY               —         */
       E$FNTF BY  00015, /* (FILE) NOT FOUND             SH,AH     */
       E$FNTS BY  00016, /* (FILE) NOT FOUND IN SEGDIR   SQ        */
       E$BNAM BY  00017, /* ILLEGAL NAME                 CA        */
       E$EXST BY  00018, /* ALREADY EXISTS               CZ        */
       E$DNTE BY  00019, /* DIRECTORY NOT EMPTY          —         */
       E$SHUT BY  00020, /* BAD SHUTDN (FAM ONLY)        BS        */
       E$DISK BY  00021, /* DISK I/O ERROR               WB        */
       E$BDAM BY  00022, /* BAD DAM FILE (FAM ONLY)      SS        */
       E$PTRM BY  00023, /* PTR MISMATCH (FAM ONLY)      PC,DC,AC  */
       E$BPAS BY  00024, /* BAD PASSWORD (FAM ONLY)      AN        */
       E$BCOD BY  00025, /* BAD CODE IN ERRVEC           —         */
       E$BTRN BY  00026, /* BAD TRUNCATE OF SEGDIR       —         */
       E$OLDP BY  00027, /* OLD PARTITION                —         */
       E$BKEY BY  00028, /* BAD KEY                      —         */
       E$BUNT BY  00029, /* BAD UNIT NUMBER              —         */
       E$BSUN BY  00030, /* BAD SEGDIR UNIT              SA        */
       E$SUNO BY  00031, /* SEGDIR UNIT NOT OPEN         —         */
       E$NMLG BY  00032, /* NAME TOO LONG                —         */
       E$SDER BY  00033, /* SEGDIR ERROR                 SQ        */
       E$BUFD BY  00034, /* BAD UFD                      —         */
       E$BFTS BY  00035, /* BUFFER TOO SMALL             —         */
       E$FITB BY  00036, /* FILE TOO BIG                 —         */
       E$NULL BY  00037, /* (NULL MESSAGE)               —         */
       E$IREM BY  00038, /* ILL REMOTE REF               —         */
       E$DVIU BY  00039, /* DEVICE IN USE                —         */
       E$RLDN BY  00040, /* REMOTE LINE DOWN             —         */
       E$FUIU BY  00041, /* ALL REMOTE UNITS IN USE      —         */
```

```
E$DNS  BY  00042, /* DEVICE NOT STARTED          —     */
E$TMUL BY  00043, /* TOO MANY UFD LEVELS         —     */
E$FBST BY  00044, /* FAM - BAD STARTUP           —     */
E$BSGN BY  00045, /* BAD SEGMENT NUMBER          —     */
E$FIFC BY  00046, /* INVALID FAM FUNCTION CODE   —     */
E$TMRU BY  00047, /* MAX REMOTE USERS EXCEEDED   —     */
E$NASS BY  00048, /* DEVICE NOT ASSIGNED         —     */
E$BFSV BY  00049, /* BAD FAM SVC                 —     */
E$SEMO BY  00050, /* SEM OVERFLOW                —     */
E$NTIM BY  00051, /* NO TIMER                    —     */
E$FABT BY  00052, /* FAM ABORT                   —     */
E$FONC BY  00053, /* FAM OP NOT COMPLETE         —     */
E$NPHA BY  00054, /* NO PHANTOMS AVAILABLE       -     */
E$ROOM BY  00055, /* NO ROOM                     —     */
E$WTPR BY  00056, /* DISK WRITE-PROTECTED        JF    */
E$ITRE BY  00057, /* ILLEGAL TREENAME            FE    */
E$FAMU BY  00058, /* FAM IN USE                  —     */
E$TMUS BY  00059, /* MAX USERS EXCEEDED          —     */
~~E$NCOM BY  00060, /* NULL_COMLINE~~                —     */
E$NFLT BY  00061, /* NO_FAULT_FR                 —     */
E$STKF BY  00062, /* BAD STACK FORMAT            —     */
E$STKS BY  00063, /* BAD STACK ON SIGNAL         —     */
E$NOON BY  00064, /* NO ON UNIT FOR CONDITION    —     */
E$CRWL BY  00065, /* BAD CRAWLOUT                —     */
E$CROV BY  00066, /* STACK OVFLO DURING CRAWLOUT —     */
E$CRUN BY  00067, /* CRAWLOUT UNWIND FAIL        —     */
~~E$CMND BY  00068, /* BAD COMMAND FORMAT~~          —     */
E$RCHR BY  00069, /* RESERVED CHARACTER          —     */
E$NEXP BY  00070, /* CANNOT EXIT TO COMMAND PROC —     */
E$BARG BY  00071, /* BAD COMMAND ARG             —     */
E$CSOV BY  00072, /* CONC STACK OVERFLOW         —     */
E$NOSG BY  00073, /* SEGMENT DOES NOT EXIST      —     */
E$TRCL BY  00074, /* TRUNCATED COMMAND LINE      —     */
E$NDMC BY  00075, /* NO SMLC DMC CHANNELS        —     */
E$DNAV BY  00076, /* DEVICE NOT AVAILABLE       DPTX   */
E$DATT BY  00077, /* DEVICE NOT ATTACHED         —     */
E$BDAT BY  00078, /* BAD DATA                    —     */
E$BLEN BY  00079, /* BAD LENGTH                  —     */
E$BDEV BY  00080, /* BAD DEVICE NUMBER           —     */
E$QLEX BY  00081, /* QUEUE LENGTH EXCEEDED       —     */
E$NBUF BY  00082, /* NO BUFFER SPACE             —     */
E$INWT BY  00083, /* INPUT WAITING               —     */
E$NINP BY  00084, /* NO INPUT AVAILABLE          —     */
E$DFD  BY  00085, /* DEVICE FORCIBLY DETACHED    —     */
E$DNC  BY  00086, /* DPTX NOT CONFIGURED         —     */
E$SICM BY  00087, /* ILLEGAL 3270 COMMAND        —     */
E$SBCF BY  00088, /* BAD 'FROM' DEVICE           —     */
E$VKBL BY  00089, /* KBD LOCKED                  —     */
E$VIA  BY  00090, /* INVALID AID BYTE            —     */
E$VICA BY  00091, /* INVALID CURSOR ADDRESS      —     */
E$VIF  BY  00092, /* INVALID FIELD               —     */
E$VFR  BY  00093, /* FIELD REQUIRED              —     */
E$VFP  BY  00094, /* FIELD PROHIBITED            —     */
E$VPFC BY  00095, /* PROTECTED FIELD CHECK       —     */
```

```
          E$VNFC BY  00096, /* NUMERIC FIELD CHECK            —        */
          E$VPEF BY  00097, /* PAST END OF FIELD              —        */
          E$VIRC BY  00098, /* INVALID READ MOD CHAR          —        */
          E$IVCM BY  00099, /* INVALID COMMAND                —        */
          E$DNCT BY  00100, /* DEVICE NOT CONNECTED           —        */
          E$BNWD BY  00101, /* BAD NO. OF WORDS               —        */
          E$SGIU BY  00102, /* SEGMENT IN USE                 —        */
          E$NESG BY  00103, /* NOT ENOUGH SEGMENTS (VINIT$)   —        */
          E$SDUP BY  00104, /* DUPLICATE SEGMENTS (VINIT$)    —        */
          E$IVWN BY  00105, /* INVALID WINDOW NUMBER          —        */
          E$WAIN BY  00106, /* WINDOW ALREADY INITIATED       —        */
          E$NMVS BY  00107, /* NO MORE VMFA SEGMENTS          —        */
          E$NMTS BY  00108, /* NO MORE TEMP SEGMENTS          —        */
          E$NDAM BY  00109, /* NOT A DAM FILE                 —        */
          E$NOVA BY  00110, /* NOT OPEN FOR VMFA              —        */
          E$NECS BY  00111, /* NOT ENOUGH CONTIGUOUS SEGMENTS          */
          E$NRCV BY  00112, /* REQUIRES RECEIVE ENABLED       —        */
          E$UNRV BY  00113, /* USER NOT RECEIVING NOW         —        */
          E$UBSY BY  00114, /* USER BUSY, PLEASE WAIT         —        */
          E$UDEF BY  00115, /* USER UNABLE TO RECEIVE MESSAGES         */
          E$UADR BY  00116, /* UNKNOWN ADDRESSEE              —        */
          E$PRTL BY  00117, /* OPERATION PARTIALLY BLOCKED    —        */
          E$NSUC BY  00118, /* OPERATION UNSUCCESSFUL         —        */
          E$NROB BY  00119, /* NO ROOM IN OUTPUT BUFFER       —        */
          E$NETE BY  00120, /* NETWORK ERROR ENCOUNTERED      —        */
          E$SHDN BY  00121, /* DISK HAS BEEN SHUT DOWN        FS       */
          E$UNOD BY  00122, /* UNKNOWN NODE NAME (PRIMENET)            */
          E$NDAT BY  00123, /* NO DATA FOUND                  —        */
          E$ENQD BY  00124, /* ENQUED ONLY                    —        */
19        E$PHNA BY  00125, /* PROTOCOL HANDLER NOT AVAIL    DPTX      */
          E$IWST BY  00126, /* E$INWT ENABLED BY CONFIG      DPTX      */
          E$BKFP BY  00127, /* BAD KEY FOR THIS PROTOCOL     DPTX      */
          E$BPRH BY  00128, /* BAD PROTOCOL HANDLER (TAT)    DPTX      */
          E$ABTI BY  00129, /* I/O ABORT IN PROGRESS         DPTX      */
          E$ILFF BY  00130, /* ILLEGAL DPTX FILE FORMAT      DPTX      */
          E$TMED BY  00131, /* TOO MANY EMULATE DEVICES      DPTX      */
          E$DANC BY  00132, /* DPTX ALREADY CONFIGURED       DPTX      */
          E$NENB BY  00133, /* REMOTE MODE NOT ENABLED        NPX      */
          E$NSLA BY  00134, /* NO NPX SLAVE AVAILABLE         ---      */
          E$PNTF BY  00135, /* PROCEDURE NOT FOUND           R$CALL    */
          E$SVAL BY  00136, /* SLAVE VALIDATION ERROR        R$CALL    */
          E$IEDI BY  00137, /* I/O error or device interrupt (GPPI)    */
          E$WMST BY  00138, /* Warm start happened (GPPI)              */
          E$DNSK BY  00139, /* A pio instruction did not skip (GPPI)   */
          E$RSNU BY  00140, /* REMOTE SYSTEM NOT UP          R$CALL    */
          E$S18E BY  00141,
     /*                                                                */
     /*   New error codes for REV 19 begin here:                       */
     /*                                                                */
          E$NFQB BY  00142, /* NO FREE QUOTA BLOCKS           —        */
          E$MXQB BY  00143, /* MAXIMUM QUOTA EXCEEDED         —        */
          E$NOQD BY  00144, /* NOT A QUOTA DISK (RUN VFIXRAT)          */
          E$QEXC BY  00145, /* SETTING QUOTA BELOW EXISTING USAGE      */
          E$IMFD BY  00146, /* Operation illegal on MFD                */
```

```
E$NACL BY  00147, /* Not an ACL directory                      */
E$PNAC BY  00148, /* Parent not an ACL directory               */
E$NTFD BY  00149, /* Not a file or directory                   */
E$IACL BY  00150, /* Entry is an ACL                           */
E$NCAT BY  00151, /* Not an access category                    */
E$LRNA BY  00152, /* Like reference not available              */
E$CPMF BY  00153, /* Category protects MFD                     */
E$ACBG BY  00154, /* ACL too big                               */
E$ACNF BY  00155, /* Access category not found                 */
E$LRNF BY  00156, /* Like reference not found                  */
E$BACL BY  00157, /* BAD ACL                                   */
E$BVER BY  00158, /* BAD VERSION                               */
E$NINF BY  00159, /* NO INFORMATION                            */
E$CATF BY  00160, /* Access category found (Ac$rvt)            */
E$ADRF BY  00161, /* ACL directory found (Ac$rvt)              */
E$NVAL BY  00162, /* Validation error (nlogin)                 */
E$LOGO BY  00163, /* Logout (code for fatal$)                  */
E$NUTP BY  00164, /* No unit table available. (PHANT$)         */
E$UTAR BY  00165, /* Unit table already returned. (UTDALC)     */
E$UNIU BY  00166, /* Unit table not in use. (RTUTBL)           */
E$NFUT BY  00167, /* No free unit table. (GTUTBL)              */
E$UAHU BY  00168, /* User already has unit table. (UTALOC)     */
E$PANF BY  00169, /* Priority ACL not found.                   */
E$MISA BY  00170, /* Missing argument to command.              */     19
E$SCCM BY  00171, /* System console command only.              */
E$BRPA BY  00172, /* Bad remote password            R$CALL     */
E$DTNS BY  00173, /* Date and time not set yet.                */
E$SPND BY  00174, /* REMOTE PROCEDURE CALL STILL PENDING       */
E$BCFG BY  00175, /* NETWORK CONFIGURATION MISMATCH            */
E$BMOD BY  00176, /* Illegal access mode   (AC$SET)            */
E$BID  BY  00177, /* Illegal identifier    (AC$SET)            */
E$ST19 BY  00178, /* Operation illegal on pre-19 disk          */
E$CTPR BY  00179, /* Object is category-protected (Ac$chg)     */
E$DFPR BY  00180, /* Object is default-protected (Ac$chg)      */
E$DLPR BY  00181, /* File is delete-protected (Fil$dl)         */
E$BLUE BY  00182, /* Bad LUBTL entry                (F$IO)     */
E$NDFD BY  00183, /* No driver for device           (F$IO)     */
E$WFT  BY  00184, /* Wrong file type                (F$IO)     */
E$FDMM BY  00185, /* Format/data mismatch           (F$IO)     */
E$FER  BY  00186, /* Bad format                     (F$IO)     */
E$BDV  BY  00187, /* Bad dope vector                (F$IO)     */
E$BFOV BY  00188, /* F$IOBF overflow                (F$IO)     */
E$LAST BY  00188; /* THIS ***MUST*** BE LAST          —        */
/*                                                             */
/*  The value of E$LAST must equal the last error code.        */
/*                                                             */
/*************************************************************/
```

FILE SYSTEM ERROR-HANDLING CONVENTIONS

All the file management system routines described in Chapter 9, and most other new subroutines, employ error-handling procedures that are standard to PRIMOS subsystems. These procedures replace the older systems using ERRVEC (Appendix E) and the altrtn argument (Chapter 14).

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 called code in this guide. 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 should always be checked for 0 or nonzero to ensure that errors do not go unnoticed. An example is:

```
    CALL CREA$$ (NAME,NAMLEN,OPASS,NPASS,CODE)
    IF (CODE.NE.0) GOTO 99
```

Standard System Error Code Definitions

Standard system error codes are variables with standardized names. In all cases, 0 means no error. Any other value identifies a particular error or exceptional (not necessarily error) condition. All reference to specific code values (other than 0) should be by the standardized names in languages where they are available. For convenience, all names are defined in files in the UFD SYSCOM on Volume 1 of the master disk. They are:

	FORTRAN 77	ERRD.INS.FTN
	FORTRAN IV	ERRD.INS.FTN
	PASCAL	ERRD.INS.PASCAL
	PL1G	ERRD.INS.PL1
	PMA	ERRD.INS.PMA
19	BASIC/VM	Not available
	COBOL	Not available

These should be included in the program with $INSERT for FORTRAN and PMA, or %INCLUDE for Pascal and PL1G.

THE ERROR-HANDLING ROUTINE ERRPR$

The following routine, ERRPR$, takes advantage of this error-handling facility, as well as allowing error-handling in user-defined subroutines.

Purpose

ERRPR$ interprets a return code and, if it is nonzero, prints a standard message followed by optional user text. It is also presented in Chapter 10.

Usage

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

key — An INTEGER*2 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*2 variable containing the return code from the routine that generated the error.

text — A message to be printed following the standard error message (any data type). text is omitted by specifying both text and txtlen as 0.

txtlen — The length in characters of text (INTEGER*2).

name — The name of the program or subsystem detecting or reporting the error (any data type). name is omitted by specifying both name and namlen as 0.

namlen — The length in characters of name (INTEGER*2).

Discussion

If code is 0, no printing occurs, and ERRPR$ immediately returns to the calling program. The format of the message for nonzero 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 0, this is followed by a blank and no more than 64 characters of text. If namlen is greater than 0, 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 returned by user-defined subroutines. User-defined errors should use codes above 10000.

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,10328,'MY MESSAGE',10,0,0)
```

will print:

```
ERROR=10328.  MY MESSAGE
```

E

Error Handling for I/O Subroutines

INTRODUCTION

The following discusses obsolete error-handling procedures for the I/O subroutines. These procedures have been replaced by return codes and the subroutine ERRPR$. (See Appendix D.)

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, described further on in this appendix. 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 access through a call to GINFO or PRERR. The contents of this vector are listed later in this appendix.

If the FORTRAN 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 are entirely up to the user.

SUBROUTINES FOR ERROR HANDLING

Three subroutines are useful for setting or retrieving information in ERRVEC: ERRSET, GETERR, PRERR.

▶ ERRSET

Purpose

ERRSET sets ERRVEC, a system vector, then takes an alternate return or prints the message stored in ERRVEC and returns control to the system.

Usage

CALL ERRSET (altval, altrtn)

CALL ERRSET (altval, altrtn, messag, num)

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 — A FORTRAN label preceded by a dollar sign. If altrtn is nonzero, control goes to altrtn. If altrtn is 0, 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 0.

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).

Discussion

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 0, 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

Purpose

A user obtains ERRVEC contents through a call to GETERR.

Usage

CALL GETERR (xervec, n)

Discussion

GETERR moves *n* words from ERRVEC into *xervec*.

On an Alternate Return:		On a Normal Return:	
ERRVEC(1)	Error code	PRWFIL:	
		ERRVEC(3)	Record number
		ERRVEC(4)	Word number
ERRVEC(2)	Alternate value		
		SEARCH:	
		ERRVEC(2)	File type

▶ PRERR

Purpose

PRERR prints an error message on the user's terminal.

Usage

```
CALL PRERR
```

Example

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 program calls SEARCH and gets an alternate return. It 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 (refer to PRWFIL and SEARCH error codes below).
(3)	X X	ERRVEC(3), ERRVEC(4), and ERRVEC(5) contain a six-character filename if 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 use.
(8)	Message length	For PRIMOS supervisor use.

PRWFIL Error Codes

Code	Meaning	
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, with one of 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, with one of the following values:

Type	Meaning
0	File is SAM.
1	File is DAM.
2	Segment directory is SAM.
3	Segment directory is DAM.
4	UFD is SAM.

F
FORTRAN Internal Subroutines

INTERNAL SUBROUTINES

The following subroutines are used internally by the FORTRAN compiler. They may be of some value to the PMA user and are briefly described. For calling sequence and further information, refer to the compiler or library source listings.

Table F-1
Subroutines Internal to FORTRAN

Subroutine	Function
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$AT	FORTRAN R-mode argument transfer subroutine.

Table F-1 (continued)
Subroutines Internal to FORTRAN

Subroutine	Function
F$ATI	FORTRAN argument transfer subroutine for PROTECTED subroutine.
F$BKSP	Backspace statement processor.
F$BN	Rewind logical device specified.
F$CB	End of READ/WRITE statement.
F$CG	FORTRAN computed GOTO processor.
F$CLOS	Close statement processor.
F$DE	Decode statement processor.
F$DEX	Decode statement processor with ERR=.
F$DN	Close (END-FILE) logical device specified.
F$EN	Encode statement processor.
F$END	Endfile statement processor.
F$FN	Provide backspace function to FORTRAN runtime programs.
F$IBR	Initialize unformatted read.
F$IBW	Initialize unformatted write.
F$IFR	Initialize formatted read.
F$IFW	Initialize formatted write.
F$ILDR	Initialize list-directed read.
F$ILDW	Initialize list-directed write.
F$INQF	Inquire by file-statement processor.
F$INQU	Inquire by unit-statement processor.
F$INR	Initialize namelist read.
F$IO77	Read and write variable-length records in default case of F$IO.

Table F-1 (continued)
Subroutines Internal to FORTRAN

Subroutine	Function
F$IOBF	F$IO buffer definition (up to 128 words, for R-mode and nonshared V-mode; up to 16K-1 words in shared V-mode library).
F$IOFTN	Read and write records in manner compatible with F$IO.
F$OPEN	Open statement processor.
F$PAUS	Pause statement processor.
F$RA	Read ASCII, no alternate returns.
F$RAX	Read ASCII, with ERR= and END= alternate returns.
F$RB	Read BINARY, no alternate returns.
F$RBX	Read BINARY with ERR= and END= alternate returns.
F$REW	Rewind statement processor.
F$RN	Read with no alternate returns.
F$RNX	Read with ERR= and END= alternate returns.
F$RTE	FORTRAN RETURN statement processor.
F$RX	COMMON read handler.
F$STOP	Stop statement processor.
F$TR	Perform the function of the FORTRAN TRACE routine.
F$WA	Write ASCII, no alternate returns.
F$WAX	Write ASCII with ERR= and END= alternate returns.
F$WB	Write BINARY, no alternate returns.
F$WBX	Write BINARY, with ERR= and END= alternate returns.
F$WN	Write with no alternate returns.
F$WNX	Write with ERR= alternate return.
F$WX	COMMON write handler.

INTRINSIC FUNCTIONS

The following subroutines are the FORTRAN library intrinsic function handlers:

Subroutine	Function
F$LS	Left shift
F$LT	Left truncate
F$OR	Inclusive OR
F$RS	Right shift
F$RT	Right truncate
F$SH	General shift

FLOATING-POINT EXCEPTIONS

The FLEX (or F$FLEX) subroutine is invoked by the compiler or system. This subroutine is the floating-point exception-interrupt processor. It determines the exception type, and returns a message as follows:

DE	Exponent underflow, store exception
DZ	Divide by 0
RI	Real-integer exception
SE	Exponent overflow

For further information on floating-point exception (FLEX), refer to the System Architecture Reference Guide.

G
Arithmetic Routines Callable from PMA

INTRODUCTION

Calls to the routines that 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.

FORMAT AND ARGUMENTS

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 G-1.) argument-1 is a number specifying the register in which the first argument is stored. (See Table G-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 G-2.)

Table G-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 (modulus)	2
SI	Magnitude of first times sign of second	2

Table G-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

Keys

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

Note

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

Note

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

Examples of format are:

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

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

A complete list of subroutines of this type follows. In the rest of this appendix, the discussion is divided into subroutines with one argument and subroutines with two arguments.

A$21
A$51
A$52
A$55
A$61
A$62
A$77

C$12
C$15
C$16
C$21
C$21G
C$25

C$26
C$27
C$51
C$52
C$57
C$61
C$62
C$67
C$75
C$76
C$77

D$21
D$27

D$51
D$52
D$55
D$57
D$61
D$62
D$67
D$71
D$77

E$11
E$21
E$22
E$26

E$27
E$51
E$52
E$55
E$57
E$61
E$62
E$66
E$67
E$71
E$77

F$CL

F$DI11
F$DI71
F$DI77

F$MA11
F$MA22
F$MA77

F$MI11
F$MI22
F$MI77
F$MO71
F$MO77

F$SI11
F$SI71
F$SI77

H$55

L$55

M$21
M$51
M$52
M$55
M$61
M$62

M$77

N$55
N$77

S$21
S$51
S$52
S$55
S$61
S$62
S$77

Z$80

SINGLE-ARGUMENT SUBROUTINES

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 0). See Table G-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 G-3.)

Zeroing

▶ Z$80

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

Table G-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	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

Keys

n/a	Not applicable
R	Used in R-mode only
R,V	Used in R- or V-modes
x	Argument type (See Table G-2.)
y	Result type (See Table G-2.)

Notes to Table G-3

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.

TWO-ARGUMENT SUBROUTINES

These subroutines perform arithmetic operations (addition, subtraction, etc.) on two arguments. 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:

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 (no "56" or "65" subroutines). The result of a two-argument subroutine is stored in the appropriate register for its data type. (See Table G-2.) For example:

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$MI11
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 G-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 G-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 G-4.

Note

In all modes, zero to the zero 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 G-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 G-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 0, the register is cleared. See Table G-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 G-5 for a complete list.

Table G-4

Two-argument
Arithmetic Subroutines (First Group)

		A$	S$	M$	D$	E$
x	y	Addition	Subtraction	Multiplication	Division	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)

Keys

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)

Note

1. Exit mode is DBL (R-mode).

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 G-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 G-5 for a complete list.

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 G-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. For example:

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

Table G-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

Keys

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)

H
SVC Information

SVCS CALLED BY PRIMOS SUBROUTINES

This Appendix defines SVCs called by PRIMOS subroutines. They are all described in this guide unless otherwise noted. SVC numbers used by PRIMOS are listed in Table H-1.

SVC INTERFACE FOR I/O CALLS

The I/O subroutines described in Chapter 16 interface with the operating system by means of supervisor call instructions (SVCs). 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 RRECL and WRECL in Chapter 17. For information about the SVC instruction, refer to the Assembly Language Programmer's Guide.

Table H-1
SVC Numbers Used by PRIMOS

Number	Associated Call
	AC$CAT (object-path, category-name, code)
	AC$CHG (name, acl-ptr, code)
	AC$DFT (name, code)
	AC$LST (name, acl-ptr, max-entries, acl-name, acl-type, code)
	AC$SET (key,name, acl-ptr, code)
	APSFX (in-pathname, out-pathname, suffix, status)
	ASNLN$ (key,line,protocol,config,lword,status)
*1500	ATCH$$ (ufdnam,namlen,ldisk,passwd,(key code))
!1400	ATTAC$ (ufdnam,namlen,ldisk,passwd,(key,loc (code)))
0100	ATTACH (ufdnam,ldisk,paswd,(key,altrtn))
*0507	BREAK$ (offon)
*0601	C1IN (char)
	CALAC$ (name, id-ptr, acess-needed, access-gotten, code)
	CAT$DL (name, code)
0602	CMREAD (char)
*1515	CNAM$$ (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$$ (ufdnam,namlen,opass,npass,code)
!1401	CREAT$ (ufdnam,namlen,opass,npass,loc(code))
0506	D$INIT (pdev)
	DIR$RD (key, unit, return-ptr, max-return-len, code)
!0410	DISCON (lun,data,statv)
*0705	DUPLX$ (key)
	ENT$RD (unit, name, return-ptr, max-return-len, code)
*1524	ERKL$$ (key,erasec,killc,code)
*1402	ERRPR$ (key,code),text,txtlen,name,namlen)
!0106	ERRTN (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)
	GETID$ (if-ptr, max-groups, code)
0112	GINFO (buff,nw)
*1504	GPAS$$ (ufdnam,namlen,opass,npass,code)
!1404	GPASS$ (ufdnam,namlen,opass,npass,code)
	GPATH$ (key, funit, buffer, bufflen, pathlen, code)
	ISACL$ (unit, code)
	NAMEQ$ (filnam1, namlen1, filnam2, namlen2)

Table H-1 (continued)
SVC Numbers Used by PRIMOS

Number	Associated Call
!0412	NETLNK (statv)
!0406	NETWAT
!0407	NTSTAT (key,p1,p2,array)
	PA$DEL (partition-name, code)
	PA$LST (name, acl-ptr, max-entries, code)
	PA$SET (partition-name, acl-ptr, code)
0111	PRERR
*1506	PRWF$$ (key,Funit,loc(bf),bflen,pos32,rnw,code)
0300	PRWFIL (key,unit,loc(buff),n,pos,altrtn)
!1406	PRWFL$ (key,unit,loc(buff),nw,pos,rnw,loc(code))
	Q$READ (buf, buflen, type, code)
	Q$SET (key, ufdnam, namlen, amount, code)
*1507	RDEN$$ (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 (semmun,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)

Table H-1 (continued)
SVC Numbers Used by PRIMOS

Number	Associated Call
	SRSFX$ (key, pathname, unit, type, n-suffixed, suffix-list, basename, suffix-used, status)
*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$SLC0 (key,line,loc(buff),nw)
*0502	TIMDAT (buff,buflen)
*0702	TNOU (msg,charcnt)
*0703	TNOUA (msg,charcnt)
!0405	TRNMIT (lun,loc(buff),cnt,statv)
	TSRC$$ (ation+newfil, pathname, funit, chrpos, type, code)
	UPDATE
!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)

Keys

* = Also direct entrance call
! = Not described in this guide

Magnetic Tape

MPC Line Printer

Output to the parallel interface line printer is accomplished through SVC calls. Refer to T$LMPC in Chapter 19.

MPC Card Reader

Input from the parallel interface card reader is controlled through SVC calls. Refer to T$CMPC in Chapter 19.

OPERATING SYSTEM RESPONSE TO SVCS

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
5-10	SVC class
11-16	SVC subclass

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 subclass are not recognized, returns to the caller at the location following the SVC code word.

The four legal syntaxes are:

1.

```
 .
 .
 .
SVC
OCT    00xxyy
DAC
DAC
 .
 .
 .
OCT    0
```

2.

```
Ent  DAC  **
     SVC
     OCT  10xxyy
```

3.

```
 .
 .
 .
SVC
OCT  04xxyy
(return-to-sender location)
DAC
DAC
 .
```

```
        .
        .
        OCT  0

    4.
        Ent  DAC   **
             SVC
             OCT  14xxyy
             (return-to-sender location)
               .
               .
               .
```

In all cases above:

xx = 6-bit class

yy = 6-bit subclass

The following classes are currently assigned:

0	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
10	Mag tape
11	Line printer
12	Card reader/punch
13	SMLC
77	Reserved for customer use

I
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 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 be concerned with the allocation process, other than to know that it works.

The second goal, referencing files by name, means selecting the desired file by giving the File Management System a 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 so that a file directory may contain names not only of files, but also of other file directories. Thus, each user may divide files into appropriate groups and subgroups as convenient.

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.

Note

For Access Control List (ACLs) protection, with Rev. 19 and higher, see the Prime User's Guide.

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 as follows: 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 0 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 0. 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 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. 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 into high-speed memory from the file starting at the file pointer, 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 (a threaded list of physical disk records). See Figure I-1.

```
                 0                 1                 n
     ___________       ___________       ___________
    |           |     |           |     |           |
    |           |     |           |     |           |
    |           |<--->|           |<--->|           |
    |           |     |           |     |           |
     ___________       ___________       ___________
      BEGINNING
        RECORD
```

SAM File Structure
Figure I-1

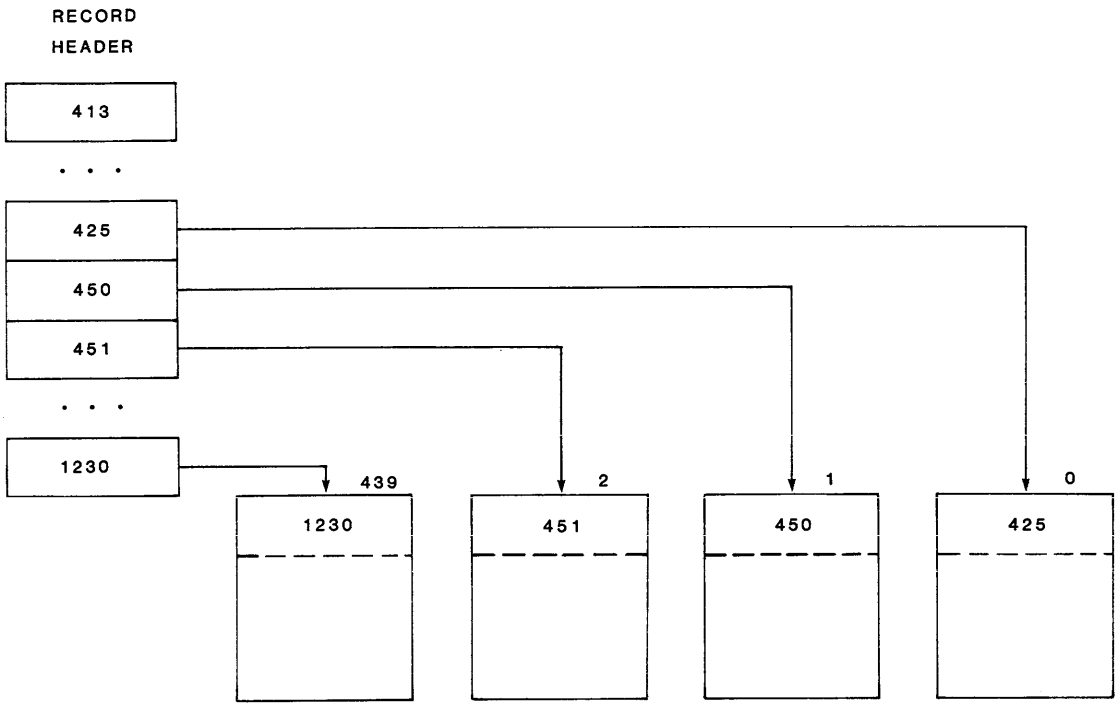
A SAM file is a collection of disk records chained together by forward and backward pointers to and from each record. Each record in a SAM file (or any file) contains a pointer to the Beginning Record Address (BRA) of the file. The first record has a pointer to the directory in which this file is an entry (root or parent 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 0 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 I-2.



DAM File Structure
Figure I-2

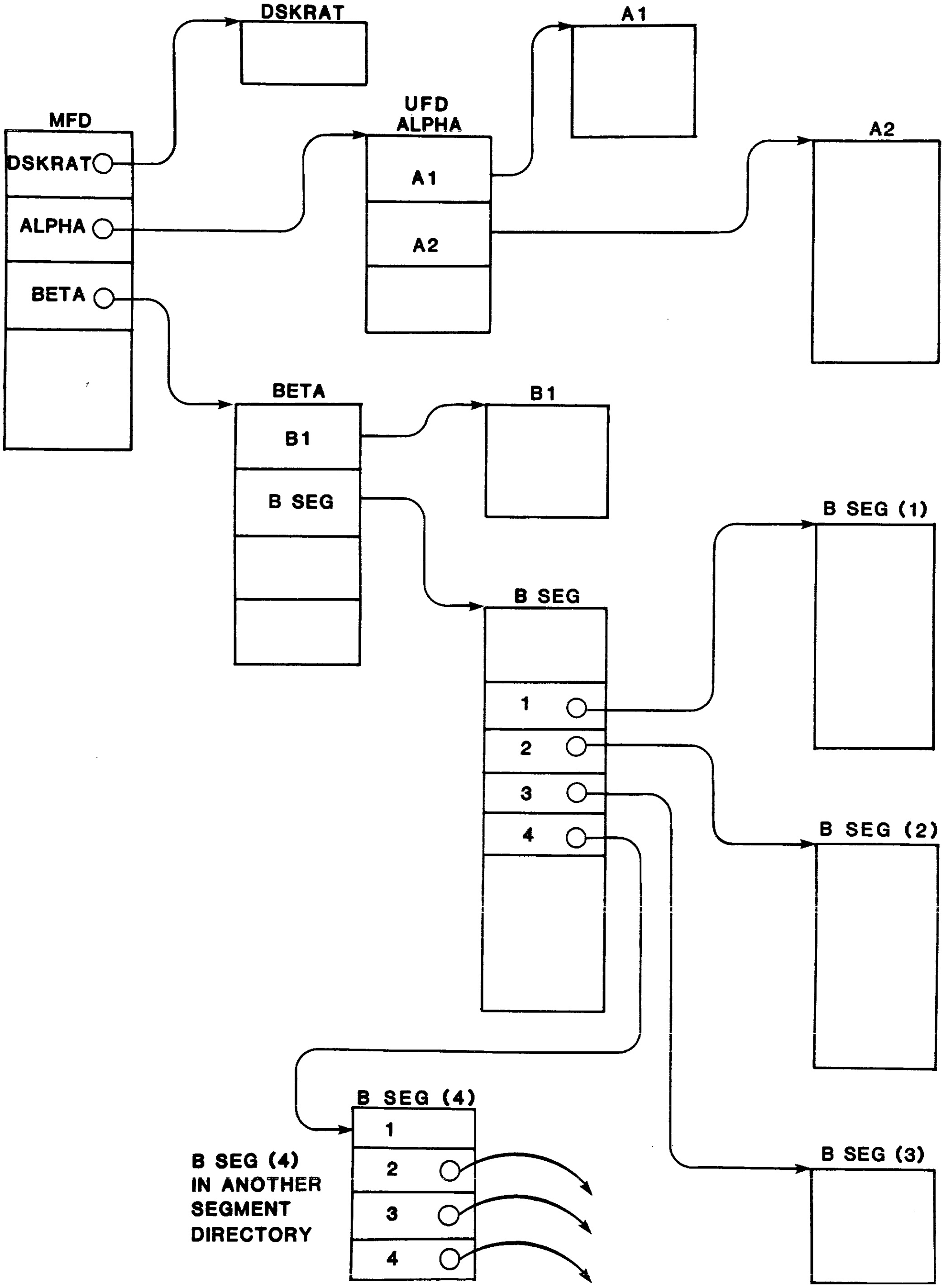
A DAM file index can exceed 512 entries on a storage module (220 entries for other devices). A multilevel index is maintained so that any record in the file can be directly accessed. (See Chapter 5 for a DAM file creation example.)

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

Record Formats

All files on PRIMOS disks are stored in fixed-length 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 16 words of the record make up the record header. Specific content of record headers is discussed later in this appendix. All



Hypothetical PRIMOS File Hierarchy with SAM and DAM File Structures

Figure I-3

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 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 card reader, paper-tape reader, etc. Each record is followed by a word containing a NEWLINE 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 LINEFEED 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 below 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 I-3 illustrates this concept. Directories are either User File Directories (UFD's) or segment directories. Each disk pack (or device, in the case of nonremovable 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 UFDs in one fundamental respect: they contain file locations but not filenames. 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, such 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 I-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 two words (INTEGER*4) that contain the address of the first physical record of the file (called the beginning record address or BRA). (See below 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 for some special 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, or a unit chosen by PRIMOS if the user specifies no unit number. 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 than 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 0 to 65535) which is the pointer to the beginning record of a file. Segment directories are therefore limited to 65536 ('200000) 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 Chapter 5 for an example of creating segment directories.

Segment directories are limited to 64K words (32K entries).

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 parent 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 '0' 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 below.

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 0 is cylinder 0, head 0, and sector 0 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 the 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 subroutines whose names begin with AT$ (Appendix A).

File Access Control

Note

For Rev. 19 and higher, see the chapter on Access Control Lists (ACLs) in the Prime User's Guide. 19

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 overwrite 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] [control-args]
```

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

Key	Value
NIL	No access allowed.
R	Read access only.
W	Write access only.
RW	Read and write access.
D	Delete only.
RD	Delete and read.

19

WD	Delete and write.
19 RWD	All access allowed (read/write/delete).

The control-args may be -REPORT or -RPT. Both specify that PRIMOS will report the results of each successful operation.

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 privileges with the segment directory.

The default protection keys of a newly created file are:

Key	Value
19 RWD	Owner has all rights.
NIL	Nonowner has none.

The passwords of a newly created UFD are:

Owner password is blank.

Nonowner password is 0. (Any password will match.)

A nonowner cannot create a new file in a UFD, or successfully give the CNAME, PASSWD, or PROTECT commands. A nonowner cannot open a current UFD for reading or writing. (See the attach commands, Appendix A, 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 appendix. 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$$) provides the ability to position the pointer anywhere within an open file. File data can be transferred anywhere in the addressing range. When a file is closed and reopened, the pointer is automatically returned to the beginning of the file. The pointer can be controlled by both the FORTRAN REWIND statement and PRWF$$ 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 appendix to allow fast random positioning. User calls to manipulate SAM and DAM files are identical.

COMMAND FILES

Note

For Rev. 19 and higher, the Command Procedure Language (CPL) is a more flexible alternative to command files. See the Prime User's Guide and the CPL User's Guide. 19

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, CMDNC0).

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 0 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$$ or CL$PIX subroutine. To add new external commands, the user simply files a memory image program (SAVEd file) under the command directory UFD (CMDNC0). Memory image files may also be kept in other directories and executed by the RESUME command.

Using 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.

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 (FIX_DISK)

19 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, FIX_DISK. For details and examples, refer to the FIX_DISK description in the System Administrator's Guide.

INTERNAL FILE FORMATS (BEFORE REV. 19)

The internal formats of all disk records in the file management system are described below with Figures I-4 through I-10. 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 preceded by a colon (:) are octal, otherwise they are decimal.

DSKRAT Format

0	8	Number Words in Header = 8
1	RECSIZ	Record Size
2	NMRECS	Number of Records in Partition (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)
	....	

DSKRAT Format
Figure I-4

RECORD HEADER FORMATS

The format of a record header is a function of the physical record size.

Record Header Format — 448-Word Records

0	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 1
Figure I-5

Record Header Format — 1040-Word Records

0	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)
10	REKLVL	Index Level for New Partition DAM Files
11		
	RESERVED	Reserved (Five Words)
15		

Record Header Format 2
Figure I-6

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 0 if it is the last record in the file.

4. REKBPT contains the address of the previous record in sequence or 0 if it is the first record in the file.

5. REKTYP is valid only in the first record of a file. Legal values are:

0	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 0 bootstrap (BOOT) or the disk record availability table (DSKRAT or volume name) and the disk has a 1040-record size (storage module), bit 1 (:100000) of REKTYP will be set.

7. DAM files on new partitions are organized somewhat differently from the above.

UFD HEADER AND ENTRY FORMATS

UFD Header Format

```
      ________
 0  |   ECW    |  ECW (See note 1 after Figure I-8.)
 1  |  OPASSW  |  Owner Password (Three Words)
    |          |
    |__________|
 4  |  NPASSW  |  Nonowner Password (Three Words)
    |          |
    |__________|
 7  |          |
    |          |
    | RESERVED |  Reserved (Sixteen Words)
    |          |
23  |__________|
```

UFD Header Format
Figure I-7

UFD Entry Format

```
    _________
 0 |   ECW    |  Entry Control Word (Type/Length)(note 1)
 1 |   BRA    |  Beginning Record Address (Two Words)
   |__________|
 3 |RESERVED  |  Reserved (Three Words)
   |          |
   |__________|
 6 |  PROTEC  |  Protection (Owner/Nonowner)(note 2)
 7 |RESERVED  |  Reserved For Future Use
 8 |  DATMOD  |  Date Last Modified (YYYYYYYMMMMDDDDD)
 9 |  TIMMOD  |  Time Last Modified (Seconds-Since-Midnight/4)
10 |  FILTYP  |  Filetype (note 3)
11 |   SCW    |  Subentry Control Word For Filename (note 4)
12 |F         |
   |  I       |
   |    L     |
   |      E   |
   |   ...    |  Filename (1-16 Words, Blank-Padded)
   |N         |
   |  A       |
   |    M     |
 N |______E___|
```

UFD Entry Format
Figure I-8

Notes

1. The Entry Control Word (ECW) consists of two 8-bit subfields. The top eight bits indicate the type of the following entry as follows:

Bit	Meaning When Bit Is On
0	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 including the ECW itself.

2. The bits in PROTEC are stored in true form (0 = no right) for both owner and nonowner fields.

3. The file type field is as before (see Record Header Format) with the 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 8-bit subfields. The top eight bits are 0, indicating subentry type 0. The low-order eight 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 FORMAT

0	BRA0	BRA of First File in Directory (Two Words)
2	BRA1	BRA of Second File in Directory (Two Words)
	0000 0000	Null Entry (Two Words)
	....	
2n	BRAn	BRA of Last File in Directory (Two Words)

Segment Directory Format
Figure I-9

DAM FILE ORGANIZATION

In old-style DAM files, the first physical record of the file was reserved to be an index to the first 440 or 1024 (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 0-439 can be accessed directly. Records 440 and above must be searched for sequentially starting with record 439.

```
  Index       Data Records
   _____
  | BRA0 |---> Record 0
  | BRA1 |---> Record 1
  |      |
  | ...  |
  |      |
  | B439 |---> Record 439---> Record 440---> Record 441---> ...
```

Old DAM Format
Figure I-10

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 multilevel 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 multilevel 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.

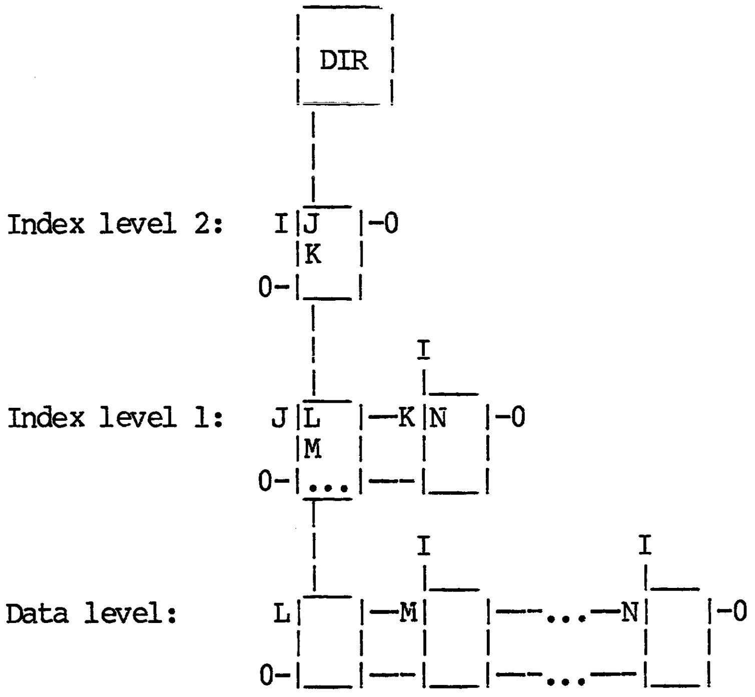
2. The index entries from the full index record are copied into one of the other new records. This record is to become the first index record of the new index level.

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 root pointers are set up as shown in the Figure I-11 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.



Keys

DIR = UFD or Segment directory
0 = Null pointer
I = Root pointer

New DAM Format
Figure I-11

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.)

J

Obsolete 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

Purpose

DISPLY updates the sense light settings according to argument Al.

The bit values of A1 (1=on, 0=off) correspond to switch/light settings which are displayed on the computer control panel.

Usage

CALL DISPLY (A1)

▶ OVERFL

Purpose

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.

Usage

CALL OVERFL (A1)

▶ SLITE

Purpose

Sets the sense light specified in argument A1 on or sets all sense lights off. If A1=0, all sense lights are reset off.

Usage

```
CALL SLITE (A1)
CALL SLITE (0)
```

▶ SLITET

Purpose

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.

Usage

```
CALL SLITET (A1,R)
```

▶ SSWTCH

Purpose

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.

Usage

```
CALL SSWTCH (A1,R)
```

K
Table of Subroutines by Function

File Management Functions

Open Files

Open, close, delete, or verify existence of a file on a specified file unit.	SRCH$$
Open a file anywhere in file system.	TSRC$$,SRSFX$
Read filename and open.	OPNP$A
Read filename and open or verify and delay.	OPVP$A
Open filename with verification and delay.	OPNV$A
Open supplied name.	OPEN$A
Open a scratch file.	TEMP$A

Close Files

Open, close, delete, or verify existence of a file.	SRCH$$
Close a file.	CLOS$A
Close a file anywhere in file system.	TSRC$$,SRSFX$

Delete Files

Open, close, delete, or verify existence of a file.	SRCH$$, FIL$DL
Delete a file.	DELE$A
Delete a file anywhere in file system.	TSRC$$,SRSFX$

Search for File

Open, close, delete, or verify existence of a file.	SRCH$$
Search for a file with any of a list of suffixes.	SRSFX$
Check for file existence.	EXST$A
Check for file anywhere in file system.	TSRC$$,SRSFX$

Manage File Attributes

Set or modify a file's attributes.	SATR$$

Find Open Filename

Find pathname for file unit or current home or attach point.	GPATH$
Check for file open.	UNIT$A

Compare Filenames

Compare two filenames for equivalence.	NAMEQ$

Change Filename

Change the name of a file.	CNAM$$

Manage ACLS

Add a file to an access category.	AC$CAT
Modify existing ACL .	AC$CHG
Set default protection.	AC$DFT
Read an ACL.	AC$LST
Create or replace an ACL.	AC$SET
Protect one file like another one.	AC$LIK
Calculate access available.	CALAC$
Delete an access category.	CAT$DL
Read directory entries.	DIR$RD
Read directory entry with given name.	ENT$RD
Determine a user's full id.	GETID$
Get directory type.	ISACL$
Delete a priority ACL.	PA$DEL
Read a priority ACl.	PA$LST
Add a priority ACL.	PA$SET
Convert an ACL directory to a password directory.	AC$RVT
Change login password.	CHG$PW
Create a password directory.	CREPW$
Search directories.	DIR$LS

Read or Write

Write to disk immediately.	FORCEW
Act on SAM or DAM files.	PRWF$$
Read ASCII characters from text files.	RDLIN$
Write ASCII characters from text files.	WTLIN$

Manage Passwords

Return passwords of a sub-UFD in current UFD.	GPAS$$
Set passwords of current UFD.	SPAS$$

Find User Information

Determine a user's full id.	GETID$

Manage Command Files

Switch between the user terminal and command file for input stream.	COMI$$
Switch terminal output to file or terminal.	COMO$$

Manage R-mode Files

Restore a R-mode runfile.	REST$$
Restore and execute an R-mode runfile.	RESU$$
Save an R-mode runfile.	SAVE$$

Manage UFDs

Attach by pathname.	AT$
Attach to a top-level directory on a given partition.	AT$ABS
Attach to a top-level directory on any partition.	AT$ANY
Attach to a UFD.	ATCH$$
Return to home directory.	AT$HOM
Attach to origin directory.	AT$OR
Attach relative to current directory.	AT$REL
Create a sub-UFD.	CREA$$
Create a password directory.	CREPW$
Read directory entries.	DIR$RD
Search directories.	DIR$LS
Read directory entry with a given name.	ENT$RD
Get directory type.	ISACL$
Position in or read from a UFD.	RDEN$$
Read quota information.	Q$READ
Set quota max.	Q$SET
Update current UFD (Primos II only).	UPDATE

Manage Segment Directories

Position, read, or modify in a segment directory.	SGDR$$
Delete a segment directory.	SGD$DL
Search directories.	DIR$LS

Position Files

Position to end of file.	GEND$A
Position file.	POSN$A
Return position of file.	RPOS$A
Rewind file.	RWND$A
Position files.	PRWF$$

Truncate Files

Truncate a file.	TRNC$A,PRWF$$

Scan File System

Search the file system structure.	TSRC$$,TSCN$A, SRSFX$

Manage Suffixes

Append a suffix to a pathname.	APSFX$
Search for a file with a list of suffixes.	SRSFX$

Check File System Objects for Validity

Check a filename for valid format.	FNCHK$
Check an id for valid format.	IDCHK$
Check a login password for valid format.	PWCHK$
Check a filename for valid format.	TEXTO$
Check a pathname for valid format.	TNCHK$,TREE$A

Prompt for and check pathname for valid format.	RNAM$A

Other PRIMOS Functions

Phantom Management

Start a phantom (obsolete).	PHANT$
Start a phantom (same login name only).	PHNTM$
Enable or disable logout notification.	LON$CN
Retrieve logout notification information.	LON$R

Read or Write

Get one character from command file or terminal.	C1IN$
Read a line of text from command file or terminal.	CL$GET
Move characters.	CNIN
Read a line of text.	COMANL
Get a character from an array.	GCHAR
Store a character in an array.	SCHAR

Error Checking

Interpret a return code.	ERRPR$

Manage User Environment

Inhibit or enable CONTROL-P.	BREAK$
Return terminal configuration word.	DUPLX$
Read or set erase and kill characters.	ERKL$$
Return to PRIMOS.	EXIT
Check operating system being used.	GINFO
Retrieve the value of a global variable.	GV$GET
Set the value of a global variable.	GV$SET
Log out a user or process.	LOGO$$
Pass control to next user.	RECYCL
Return system and user information.	TIMDAT
Return process number and user count.	USER$
Return type of current process.	UTYPE$
See also Date-time Routines.	

String Manipulation Routines

Compare two strings for equality.	CSTR$A
Compare two substrings for equality.	CSUB$A
Fill a string with a character.	FILL$A
Fill a substring with a given character.	FSUB$A
Get a character from a packed string.	GCHR$A
Left-justify, right-justify, or center a string.	JSTR$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

Convert between upper- and lowercase.	CASE$A
Check data type.	TYPE$A
Rotate substring left or right.	RSUB$A
Shift string left or right.	SSTR$A
Shift substring left or right.	SSUB$A
Test for pathname.	TREE$A
Determine string type.	TYPE$A

User Query Routines

Prompt and read a pathname and check for validity.	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

Mathematical Routines

Generate random number and update seed, based upon a 32-bit word size and using the linear congruential method.	RAND$A
Initialize random number generator seed.	RNDI$A

Conversion Routines

Convert a string from lower- to uppercase or upper- to lowercase.	CASE$A
Convert ASCII number to binary.	CNVA$A
Convent binary number to ASCII.	CNVB$A
Make a number printable, if possible (convert to FORTRAN format).	ENCD$A

Parsing Routine

Parse PRIMOS command line.	CMDLA,CLPIX, RDTK$$

Date-time Routines

Convert binary date to quadseconds.	CV$DQS
Convert formatted date to binary.	CV$DTF
Convert binary date ISO format.	CV$FDA
Convert binary date to visual format.	CV$FDV
Return current date/time in binary format.	DATE$
Return time, date, and other information.	TIMDAT
Convert the DATMOD field (as returned by RDEN$$) to the form DAY MON DD YYYY.	FDAT$A
Convert the DATMOD field (as returned by RDEN$$) to the form DAY DD MON YYYY.	FEDT$A
Convert the TIMMOD field (as returned by RDEN$$).	FTIM$A
CPU time since login.	CTIM$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

Matrix Operations

Operation		Integer	Single Precision	Complex	Double Precision
Setting matrix to identity 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	CMSUB	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

Sort, Merge, and Search Routines

Sort one ASCII file on one ASCII key.	SUBSRT
Sort/merge sorted files (multiple key type).	ASCSRT,ASCS$$
Merge sorted files.	MRG1$S
Return next merged record to sort.	MRG2$S
Close merged input files.	MRG3$S
Sort several input files.	SRTF$S
Prepare sort table + buffers.	SETU$S
Get input records.	RLSE$S
Sort tables prepared by SETU$S.	CMBN$S
Get sorted records.	RTRN$S
Close all sort units.	CLNU$S
Heap sort.	HEAP
Partition exchange.	QUICK
Diminishing increment.	SHELL
Radix exchange.	RADXEX
Insertion sort.	INSERT
Bubble sort.	BUBBLE
Binary search/build binary table.	BNSRCH

Temporary Device Assignment

Assign device.	ATTDEV

Device-independent Drivers

Read ASCII.	RDASC
Write ASCII.	WRASC
Read binary.	RDBIN
Write binary.	WRBIN
Other control functions (obsolete).	CONTRL

Device-dependent Drivers (Peripheral Handlers)

User Terminal or Input Command Stream

Output a blank line to terminal.	TONL
Output characters with LF and CR.	TNOU
Output characters to terminal.	TNOUA
Output 16-bit integer.	TOVFD$
Read character from terminal into Register A.	T1IB
Read character from terminal into variable.	T1IN
Write character from Register A to terminal.	T1OB
Write character from variable to terminal.	T1OU
Input decimal number.	TIDEC
Input hexadecimal number.	TIHEX
Input octal number.	TIOCT
Input number in specified format.	RNUM$A
Output 6-character signed decimal number.	TODEC
Output 4-character unsigned hexadecimal number.	TOHEX
Output 6-character unsigned octal number.	TOOCT
Read ASCII from terminal.	I$AA01
Read ASCII from terminal or input stream.	I$AA12
Write ASCII to terminal or command stream.	O$AA01
Read binary from terminal.	I$BA01
Write binary to terminal.	O$BA01
Other control functions.	C$A01

Paper Tape

Input character from paper tape to Register A.	P1IB
Input character from paper tape to variable.	P1IN
Output character from the Register A to paper tape	P1OB
Output character from variable to paper tape.	P1OU
Read paper tape (ASCII).	I$AP02
Read paper tape (binary).	I$BP02
Punch paper tape (ASCII).	O$AP02
Punch paper tape (binary).	O$BP02
Other control functions.	C$P02

Disk

Read ASCII compressed.	I$AD07
Read ASCII uncompressed.	I$AD07
Write ASCII compressed.	O$AD07
Write ASCII uncompressed.	O$AD08
Read binary compressed.	I$BD07
Read binary uncompressed.	I$BD07
Write binary compressed.	O$BD07
Write binary uncompressed.	O$BD07
Other control functions.	SRCH$$

Line Printers

Centronics LP.	O$AL04
Parallel interface to line printer (MPC).	O$AL06
Vesatec printers.	O$AL14
Move data to LPC line printer.	T$LMPC
Insert a file in spooler queue.	SPOOL$

Printer/Plotter

Versatec.	TVG,OAL14

Card Reader/Punch

Input from parallel card reader.	I$AC03
Input from serial card reader.	I$AC09
Read and print card from parallel interface reader.	I$AC15
Input from MPC card reader.	T$CMPC
Parallel interface to card punch.	O$AC03
Parallel interface to card punch and print on card.	O$AC15
Raw data mover.	T$PMPC

Magnetic Tape

Control 9-track ASCII/binary.	C$M05
Control 7-track ASCII/binary.	C$M10
Control 7-track BCD.	C$M11
Control 9-track EBCDIC.	C$M13
Write ASCII to 9-track.	O$AM05
Write ASCII to 7-track.	O$AM10
Read ASCII to 9-track.	I$AM05
Read ASCII from 7-track.	I$AM10
Write binary to 9-track.	O$BM05
Write binary to 7-track.	O$BM10
Read binary from 9-track.	I$BM05
Read binary from 7-track.	I$BM10
Write BCD to 7-track.	O$AM11
Write EBCDIC to 9-track.	O$AM13
Read BCD from 7-track.	I$AM11
Read EBCDIC from 9-track.	I$AM13
Raw data mover.	T$MT

Communications Handlers

Communicate with SMLC driver.	T$SLC0
Assign AMLC line.	ASNLN$
Communicate with AMLC driver.	T$AMLC

Semaphore-handling Subroutines

Open (request) semaphore:	
by filename.	SEM$OP**
by file unit.	SEM$OU**
Notify semaphore.	SEM$NF
Wait.	SEM$WT
Test counter.	SEM$TS
Drain (reset counter or notify).	SEM$DR

Set timer.	SEM$TN*
Timed wait.	SEM$TW**
Close semaphore.	SEM$CL**
Suspend process.	SLEEP$

*For numbered semaphores only
**For named semaphores only

Condition-mechanism Subroutines

Call more on-units.	CNSIG$

Action	Programming Language			
	FTN	F77	PL1G	PMA
Create an on-unit.	MKON$F	MKON$P	MKON$P	MKONU$
Signal a condition.	SGNL$F	SGNL$F	SIGNL$	SIGNL$
Cancel (revert) an on-unit.	RVON$F	RVON$F	RVONU$	RVONU$
Nonlocal GOTO.	PL1$NL	PL1$NL	(1)	PL1$NL
Make PL/I-compatible label.	MKLB$F	MKLB$F	(1)	MKLB$F

Note

1. Supported directly by the programming language.

Message Subroutines

Send a message to another user.	SMSG$
Receive a deferred mesage.	RMSGD$
Set receiving state for messages.	MGSET$
Return receiving state of a user.	MSG$ST

L
EPF Subroutines

INTRODUCTION

With the release of Revision 19.4, the two utilities for linking and loading programs (LOAD and SEG) are now augmented by a new linker: BIND. Instead of creating programs that must execute in the same portion of memory each time they are run, BIND creates Executable Program Formats (EPFs). EPFs make it easier for you to build and maintain software. EPFs can maximize a virtual memory system, because PRIMOS takes care of address space allocation at program load time instead of build time (as with SEG and LOAD).

For a description of the use and the advantages of this new utility, refer to the Programmer's Guide to BIND and EPFs. For a thorough discussion of how to fine-tune your system using EPFs, and why you will want to do so, refer to the Advanced Programmer's Guide.

This appendix contains those subroutines that support an EPF-based environment. They let you use the new features associated with EPFs, or let you transform older programs into EPFs without making any internal programming changes.

Subroutine	Function
ALC$RA	Allocate (process-class) space for return function data.
ALS$RA	Allocate (process-class) space for return function data and set its value.

CE$BRD Return the command environment breadth allocated to the user.

CE$DPT Return the command environment depth allocated to the user.

CKDYN$ Determine Primos' runtime accessibility to an entrypoint via a dynamic link (Dynt).

CP$ Invoke a command or program from within a running program.

DY$SGS Retrieve the maximum number of private dynamic segments.

EPF$ALLC Allocate storage for an EPF's linkage and static data areas.

EPF$AL As for EPF$ALLC above, but used for FTN calls.

EPF$CPF Return the state of the command processor flags in an EPF.

EPF$CP As for EPF$CPF above, but used for FTN calls.

EPF$DEL Deactivate one activation of an EPF for the calling process.

EPF$DL As for EPF$DEL above, but used for FTN calls.

EPF$INIT Perform linkage initialization for an EPF.

EPF$NT As for EPF$INIT above, but used for FTN calls.

EPF$INVK Begin the actual execution of a program EPF.

EPF$VK As for EPF$INVK above, but used for FTN calls.

EPF$MAP Map the procedure images of an EPF file into virtual memory.

EPF$MP As for EPF$MAP above, but used for Fortran calls.

EPF$RUN Perform the appropriate calls to execute an EPF file.

EPF$RN As for EPF$RUN above, but used for FTN calls.

EX$CLR Disable the signalling of the EXIT$ condition.

EX$RD Return the state of the counter controlling the EXIT$ condition.

EX$SET Enable the signalling of the Exit$ condition.

FRE$RA Release space designated for EPFs returning information via command functions.

ICE$ Initialize the command environment.

RD$CE_DP Return the current value of the command environment breadth.

RD$CED AS for RD$CE_DP above, but used for FTN calls.

RPL$ Replace an EPF file with another one.

STR$AL Allocate space in user-class storage and return an error code to caller.

STR$AP Allocate space in process-class storage.

STR$AS Allocate space in subsystem-class storage.

STR$AU Allocate space in user-class storage.

STR$FP Free space from process-class storage.

STR$FR Free space from user-class storage and return an error code to the caller.

STR$FS Free space from subsystem-class storage and return an error code to the caller.

STR$FU Free space from user-class storage.

ST$SGS Retrieve the maximum number of private static segments.

Note

The following subroutine descriptions use a PL/I-like format to supply a base for consistency in the presentation of data structures.

▶ ALC$RA

Purpose

This routine allocates space for EPF function return information. Refer to the Advanced Programmer's Guide for a complete discussion of ALC$RA and ALS$RA.

When a function returns information, it passes the data to the caller via an assignment statement. For an EPF to do this, it must create an indirect pointer and a storage area, so that when the data is returned at execution time it can be stored and accessed by the caller of the function. In order to pass such information to the operating system, an interface (given in the discussion below) defines rtn_fcn_ptr and rtn_fcn_struc.

ALC$RA provides you the space for rtn_fcn_struc; it also returns the value for rtn_fcn_ptr which you can then pass back to the caller of the EPF function.

Note

> Because this interface requires the caller to perform pointer-based operations, the caller should be a PMA or PL/I Subset G program. Other programs should use the ALS$RA subroutine.

Usage

```
dcl alc$ra entry (fixed bin(31), ptr);

call alc$ra(space_needed, rtn_fcn_ptr);
```

space_needed (INPUT) — The total amount of space needed for rtn_fcn_struc (in 16-bit halfwords). It is the sum of the space needed for the return value and the rtn_fcn_struc version number.

rtn_fcn_ptr (OUTPUT) — The pointer to the information to be returned by the function.

Discussion

Refer to chapters 18 and 19 of the Advanced Programmer's Guide for a detailed discussion of the following interface.

When using dynamic storage allocation, an EPF program acting as a function (that is, passing back some result to the operating system) must first have the following interface defined:

```
dcl your_epf entry(char(1024) var, fixed bin(15),
                   1, 2 char(32) var,
                      2 fixed bin (15),
                      2 ptr,
                      2, 3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(11),
                         3 bit(1),
                         3 bit(1),
                         3 bit(14),
                         3 fixed bin(15),
                         3 fixed bin(15),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(13),
                   1, 2 bit(1),
                      2 bit(15),
                   ptr);

call your_epf(command_args, command_status, command_state,
              command_fcn_flags, rtn_fcn_ptr);
```

These arguments are defined as follows:

command_args — The entire command line as entered by user

command_status — The command status returned by the program to the operating system:

= 0 No error.
> 0 Fatal error.
< 0 Soft error or warning.

command_state — Information relative to this invocation. It contains, in the order specified:

command name — command entered by user.

version — current version of the structure of the command state (1 at Rev. 19.4).

vcb_ptr — pointer to CPL local variables.

preprocessing_info — information relating to what has been preprocessed:

mod_after_date — if nonzero, then the command processor has found something modified after the given date.

mod_before_date — if nonzero, then the command processor has found something modified before the given date.

bk_after_date — if nonzero, then the command processor has found something backed up after the given date.

bk_before_date — if nonzero, then the command processor has found something backed up before the given date.

type_dir — a directory has been found that matches a wildcard.

type_segdir — a segment directory has been found that matches a wildcard.

type_file — a file has been found that matches a wildcard.

type_acat — an access category has been found that matches a wildcard.

type_rbf — a ROAM based file has been found that matches a wildcard.

res1 — 11 bits that are undefined.

verify_sw — the -VERIFY option has been given.

botup_sw — perform full treewalk before executing program.

res2 — 14 bits that are undefined.

walk_from — the tree level at which the present treewalk started.

walk_to — the present treewalk level.

in_iteration — command processor is currently in an iteration sequence.

in_wildcard — command processor is currently in a wildcard sequence.

in_treewalk — command processor is currently in a treewalk sequence.

res3 — 13 bits that are undefined.

command_fcn_flags — information relative to this command function invocation. Its contents in the order specified are:

command_fcn_call — indicates that this program has been called as a command function.

reserved — 15 bits that are undefined.

rtn_fcn_ptr — pointer to a structure that describes the values returned to the caller of the EPF function. This structure is itself defined as:

```
dcl 1 rtn_fcn_struc,
        2 version fixed bin(15),
        2 value_str char(*) var;
```

Where:

version — is the structure's version (see ensuing discussion).

value_str — is a string of 1 to 32767 (or any language-imposed limit) characters holding the value to be returned.

First obtain the value of rtn_fcn_ptr by calling ALC$RA (or ALS$RA). After the call to ALC$RA, your program must set the version number of rtn_fcn_struc to 0 and copy the value of that structure into value_str. Then the interface sets rtn_fcn_ptr in its main entrypoint's calling sequence and returns to the calling program.

▶ ALS$RA

Purpose

This routine is used both to allocate space from process-class storage for EPF function return information and to actually set the value of the information. It also assigns the value 0 to version within the rtn_function_structure (see rtn_function_addr below).

Usage

```
dcl als$ra entry (char(*), fixed bin(31), ptr);

call als$ra (function_result_str, char_size_of_str,
             rtn_function_addr);
```

function_result_str (INPUT) — The character string that is the result of the program invoked as a function. The string may contain up to 8192 characters.

char_size_of_str (INPUT) — The number of characters in function_result_str.

rtn_function_addr (OUTPUT) — The address allocated to the rtn_fcn_struc. The return_function_structure itself has this format:

```
1 rtn_fcn_struc,
        2 version fixed bin(15),
        2 value_str char(*) var;
```

The address is returned as a pointer to the EPF function that called ALS$RA; the calling function then stores it in rtn_function_ptr for further use.

▶ CE$BRD

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the current user's profile. This routine returns the maximum number of simultaneous program EPF invocations per command level, that is, the command environment breadth allocated to the user.

Usage

```
dcl ce$brd entry () returns (fixed bin(15));

maximum_command_env_depth = ce$brd();
```

▶ CE$DPT

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the current user's profile. This routine returns the maximum number of command environment levels, that is, the command environment depth allocated to the user.

Usage

```
dcl ce$dpt entry () returns (fixed bin(15));

maximum_command_env_depth = ce$dpt();
```

▶ CKDYN$

Purpose

This routine accepts a dynamic entrypoint (DYNT) name and determines whether that routine is accessible through the Primos dynamic linking mechanism.

Usage

```
dcl routine_name entry (char(32) var, fixed bin (15));

call ckdyn$ (routine_name, code);
```

routine_name (INPUT)	The name of the dynamic entrypoint.
code (OUTPUT)	If CKDYN$ finds the dynamic entrypoint, code is reset (0). Otherwise code is returned as E$FNTF (not found).

▶ CP$

Purpose

This routine is the interface into the Primos command processor for invoking a command from a running program.

This routine should be called whenever a user wants to invoke a command or program from within a running program, and wishes to make use of the extended command processing features available from the standard command processor. Arguments that must be passed are command_line, status, and command_status; other arguments are optional.

For a thorough discussion of the use of CP$ within an EPF-based environment, refer to Chapter 19 of the Advanced Programmer's Guide.

Usage

```
dcl cp$ entry (char(1024) var, fixed bin(15), fixed bin(15),
               1, 2 bit(1), 2 bit(1), 2 bit(14),
               ptr, ptr);

call cp$ (command_line, status, command_status, command_flags,
          local_variable_ptr, rtn_function_ptr);
```

command_line (INPUT) — The actual command or program being invoked.

status (OUTPUT) — Return command invocation error status.

command_status (OUTPUT) — Return command execution error status, defined in Appendix D.

flags (INPUT) — This field contains information relative to the command function invocation. It has this format:

```
1 flags,
  2 command_function_call bit(1),
  2 no_eval_vbl_fcns bit(1),
  2 reserved bit(14);
```

The first bit, if set, indicates that the program was called as a command function; the second, if set, indicates that command function and global variable references are to be passed without modification; the remaining fourteen bits are undefined.

local_variable_ptr
(INPUT)

The pointer to local variables allocated during execution, if this CP$ call is made by a program executed from within a CPL file. (Compare this option within CP$ to the general purpose for LV$GET.)

rtn_function_ptr
(OUTPUT)

Pointer to a return_function_structure for command function processing. The return_function_structure itself has the following format.

```
1 rtn_function_structure,
     2 version fixed bin(15),
     2 char_string char(*) var;
```

Refer to the description of this and other parts of the interface structure in the discussion following ALC$RA.

Discussion

CP$ provides an easy-to-use interface to call external programs. All a programmer has to do is call CP$ with an argument that represents a command line. This pseudo command line will be a character string representation of the external program to be called. CP$ will perform all wildcarding, treewalking, and iteration in reference to the character string; however, it does not perform abbreviation expansion.

For example, a user may have a purchasing program that allows several different commands, each of which calls an external program that can be called by cp$. In this case, the purchasing program prompts the user to insert a command-line; the user inputs "ORDER wrench" (or the longer form given below). ORDER is the name of the external program that does the ordering. Part of the purchasing program would therefore resemble the following:

```
    /* At this point the User is prompted to input a command. */
    /* The User now wants to "ORDER wrench".  But, unless ORDER  */
    /* is in CMDNC0, the Resume command must be added to execute */
    /* ORDER, which is probably one of several programs within a  */
    /* subdirectory of programs: "Resume PROGS>ORDER wrench." */

    /* The subroutine cl$get is called to gather the terminal input. */

    call cl$get(command_line, command_line_length, status);

    /* Now CP$ uses that command_line to fetch */
    /* the program that will honor this request. */

    call cp$('RESUME PROGRS>ORDER wrench', status, command_status);
```

DY$SGS

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the current user's profile. This routine retrieves the maximum number of private, dynamic segments allocated to the user.

Usage

```
dcl dy$sgs entry () returns (fixed bin(15));

maximum_private_dynamic_segs = dy$sgs();
```

EPF$ALLC
(or EPF$AL for FTN calls)

Purpose

This routine performs the "linkage allocation" phase for an EPF. The storage for the linkage and static data areas of an EPF is allocated here. All the template information for the storage needs is contained within the EPF file itself.

Memory storage is allocated from temporary segments in the dynamic segment range. All storage is managed within PRIMOS. The type of storage is determined by the type of EPF. Program EPFs and program-class library EPFs are allocated storage in user-class storage. Process class library EPFs are allocated storage in process-class storage.

Usage

```
dcl epf$allc entry (fixed bin (31), fixed bin);

call epf$allc(epf_id, status);
```

epf_id (INPUT) The identifier of the mapped-in EPF (created by EPF$MAP)

status (OUTPUT) A standard success/failure code returned by the routine.

The following errors may be returned to the caller of EPF$ALLC:

E$BPAR An invalid epf_id has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

E$ILTD An invalid EPF LTD linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.

E$EPFT An invalid EPF type field was detected when trying to allocate storage. Resubmit the file to BIND.

▶ EPF$CPF
(or EPF$CP for FTN calls)

Purpose

This routine returns the state of the command processing flags in an EPF. The command processing features of the EPF are set by the generator of the EPF by using BIND, the linker used for EPFs.

Usage

```
dcl epf$cpf entry (fixed bin (31),        /* epf_id
                   1, 2, 3 bit(1),        /* epf_info
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3, 4 bit(1),
                            4 bit(1),
                            4 bit(1),
                            4 bit(1),
                            4 bit(1),
                         3 reserved bit(7),
                  2 fixed bin(15),
                fixed bin(15));        /* status

call epf$cpf (epf_id, epf_info, status);
```

epf_id (INPUT) The identifier of the mapped-in EPF.

epf_info (INPUT/OUTPUT) The structure that is to contain the return information about the command processing features of the EPF desired by the caller. Refer to the Advanced Programmer's Guide Ch. 19 for explanations of each bit. The format of the structure follows:

```
1 epf_info based,
      2 command_flags,
        3 wildcards bit(1),
        3 treewalks bit(1),
        3 iteration bit(1),
        3 verify bit(1),
        3 file_types,
          4 file bit(1),
          4 directory bit(1),
          4 segdir bit(1),
          4 acat bit(1),
          4 rbf bit(1),
          4 reserved bit(7),
      2 name_generation_position fixed bin(15);
```

status (OUTPUT) Error status. The following error may be returned to the caller of EPF$CPF:

E$BPAR An undefined epf_id has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

▶ EPF$DEL
(or EPF$DL for FTN calls)

Purpose

This routine effectively deactivates one activation of an EPF for the calling process. The segment(s) used for linkage and static data for the most recent invocation of the EPF are returned to the free pool of dynamic segments. If this EPF has not been previously executed by a call to EPF$INVK, the EPF procedure segment(s) are released, and the storage used by the in-memory EPF data base is released.

The invocation of the EPF utilizes valuable system resources. Each invocation of an EPF program should be followed by a call to EPF$DEL to free the storage allocated for program linkage and static storage, unless the EPF is to be invoked shortly.

If the EPF invocation is not terminated by a call to EPF$DEL, system segments are not efficiently returned to the free pool, and a user may quickly run out of segments in the dynamic segment range.

Usage

```
dcl epf$del entry (fixed bin (31), fixed bin(15));

call epf$del (epf_id, status);
```

epf_id (INPUT) — The identifier of the EPF created by EPF$MAP. The most recent invocation of the EPF will be deactivated.

status (OUTPUT) — Return EPF invocation error code. An error may occur while attempting to return EPF procedure segments to the system. Should this happen, the user's command environment is reinitialized after the following message is displayed:

```
Unable to free
EPF procedure segments.
```

Any errors detected when de-allocating storage cause an appropriate error message to be displayed at the user's terminal and the user's command environment to be reinitialized.

The following error codes may be returned to the caller of EPF$DEL:

E$BPAR — An undefined epf_id has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

E$EPFT An invalid EPF type field was detected. Resubmit the file to BIND.

E$BVER An invalid EPF version was detected. Resubmit the file to BIND.

▶ EPF$INIT (or EPF$NT for FTN calls)

Purpose

This routine performs the "linkage initialization" phase for an EPF. The EPF must already be mapped to memory (by EPF$MAP), with its static data areas already allocated (by EPF$ALLC).

Usage

```
dcl epf$init entry (fixed bin(15),fixed bin(31), fixed bin(15));

call epf$init (key, epf_id, status);
```

key (INPUT) Is an action specifier key. Possible values are: K$INITALL (or K$INAL for FTN callers), which specifies a complete initialization of data areas; K$REINIT (or K$REIN for FTN callers), which specifies only a re-initialization of the data areas, that is, reinitialize only the static data and faulted IPs but maintain other data such as IPs and ECBs.

epf_id (INPUT) The identifier of the mapped-in EPF (created by EPF$MAP).

status (OUTPUT) Is a standard success/failure code returned by the routine. The following errors may be returned to the caller of EPF$INIT:

E$BARG Linkage and static data areas for the EPF were not allocated. Call EPF$ALLC before calling EPF$INIT.

E$BKEY An invalid key was used in the call. Resubmit the file to BIND.

E$BLTE An invalid EPF LTE linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.

E$BLTD An invalid EPF LTD linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.

E$BPAR An undefined epf_id has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

E$BVER An invalid EPF version was detected. Resubmit the file to BIND.

E$EPFT An invalid EPF type field was detected when trying to allocate storage. Resubmit the file to BIND.

▶ EPF$INVK
(or EPF$VK for FTN calls)

Purpose

This routine initiates the actual execution of a program EPF. At this point, the EPF should have been mapped into virtual memory and the static data areas should be both allocated and initialized. The order of calls should be EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK.

The address of the starting entry control block for the EPF is found from the Control Information Block (CIB) within the EPF, and the EPF is invoked by issuing a PCL instruction to the ECB.

Program EPFs written as programs (that is, they expect no command arguments and return no severity code) are normally invoked with the first calling format below. Those program EPFs written as functions, and those expecting arguments, must be invoked using the second format below. The additional arguments are provided for passing invocation information to the program being invoked, and for returning data to the invoking program.

Usage

```
dcl epf$invk entry (fixed bin (31), fixed bin(15));

call epf$invk(epf_id, status);
```

or

```
dcl epf$invk entry (fixed bin (31), fixed bin(15),
                    char(1024) var, fixed bin(15),
                    1, 2 char(32) var,
                       2 fixed bin(15),
                       2 ptr,
                       2, 3 fixed bin(31),
                          3 fixed bin(31),
                          3 fixed bin(31),
                          3 fixed bin(31),
                          3 bit(1),
                          3 bit(1),
                          3 bit(1),
                          3 bit(1),
                          3 bit(1),
                          3 bit(11),
                          3 bit(1),
                          3 bit(1),
                          3 bit(14),
                          3 fixed bin(15),
                          3 fixed bin(15),
                          3 bit(1),
                          3 bit(1),
                          3 bit(1),
                          3 bit(13),
                    1, 2 bit(1),
                       2 bit(15),
                    ptr);

call epf$invk(epf_id, status, com_args, com_status, com_state,
              flags, rtn_function_ptr);
```

epf_id (INPUT) — The identifier of the EPF (created by EPF$MAP).

status (OUTPUT) — Return EPF invocation error code. Possible error codes are:

E$BPAR — Undefined identifier of the EPF has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

E$EPFT — An invalid EPF type field was detected. Resubmit the EPF to BIND.

E$BVER — An invalid EPF version was detected. Resubmit the EPF to BIND.

com_args (INPUT)

The command arguments.

com_status (OUTPUT)

The command execution error status. The standard error codes generated during program execution may be returned. Refer to Appendix D for a complete list.

com_state (INPUT)

Contains information relative to the EPF invocation. Subdivisions of that information are as follows:

```
1 com_state,
  2 com_name,
  2 version,
  2 vcb_ptr,
  2 cp_iter_info,
    3 mod_after_date,
    3 mod_before_date,
    3 bk_after_date,
    3 bk_before_date,
    3 type_dir,
    3 type_segd,
    3 type_file,
    3 type_acat,
    3 type_rbf,
    3 verify_sw,
    3 botup_sw,
    3 walk_from,
    3 walk_to,
    3 in_iteration,
    3 in_wildcard,
    3 in_treewalk,
```

with fields that contain the following assignments:

com_name

Name of the EPF command.

version

Version of the com_state structure, set to either 0 or 1; 0 signals that only these first two fields have defined values, while 1 signals that all four of these are defined and provided by the caller.

vcb_ptr

Ptr to local CPL variables allocated during the execution of a CPL program. This field is null () if there is no invoking CPL program.

cp_iter_info — Information relative to the extended command processing features for the command. This information is passed to the program EPF from the program that is invoking it.

flags (INPUT) — This field contains information relative to the command function invocation. It has this format:

```
1 flags,
  2 command_function_call bit(1),
  2 reserved bit(15);
```

The first bit, if set, indicates that the program was called as a command function; the remaining fifteen bits are undefined.

rtn_function_ptr (OUTPUT) — Pointer to a rtn_fcn_struc that is used by an EPF acting as a function. The rtn_fcn_struc itself has this format:

```
1 rtn_fcn_struc,
   2 version  fixed bin (15),
   2 value_str  char (*) var;
```

The version must be set to zero by the EPF function.

The memory space for this data will have been allocated by the EPF. The caller of the function utilizes this data and later de-allocates the memory space by calling FRE$RA.

▶ EPF$MAP
(or EPF$MP for FTN calls)

<u>Purpose</u>

This routine is called to map the procedure images of an EPF file into virtual memory. This is the "EPF-mapping" phase of the Executable Program Format (EPF) mechanism. The EPF file should already be open for VMFA-read (K$VMR) on a file unit; that is, you must first call either SRCH$$, SRSFX$, or TSRC$$ with the K$VMR key specified.

If the EPF file in question is to be used as a program (rather than a library), then this routine is the first of four subroutines that must be called in this order: EPFMAP, EPFALLC, EPF$INIT, EPF$INVK. Refer to chapters 1 and 2 of the Advanced Programmer's Guide for more information on program and library EPFs.

Usage

```
dcl epf$map entry (fixed bin(15), fixed bin(15), fixed bin(15),
                   fixed bin(15)) returns (fixed bin (31));

epf_id = epf$map (key, unit, access_rights, status);
```

key (INPUT)

Use one of the following:

K$ANY Use any available segment(s).

K$COPY Copy the segment-image(s) of the file into temporary segment(s). DBG uses this option to obtain writable segment(s) for debugging.

K$DBG Map DBG information. Used only by DBG, this causes the segment-image(s) of the EPF file that contain the DBG information to be mapped into memory.

unit (INPUT) The file unit on which the EPF is currently open for VMFA-read, K$VMR.

access_rights (INPUT) The access rights to place on the VMFA segments. Possible values are:

K$R Read only access on segment
K$RX Read/execute access

Currently, K$R and K$RX have the same effect; use K$RX to be assured of no future compatibility problems.

status (OUTPUT) A standard success/failure code returned by the routine. The EPF must be successfully mapped to memory in order to be executed. The user code that calls EPF$MAP or EPF$RUN should be capable of dealing

	with any error condition that might result when the EPF is invoked. Refer to "Error Processing" for a treatment of possible failure codes.
epf_id (OUTPUT)	The identifier of the mapped-in EPF. This identifier should be used as a handle to the EPF in memory when calling the remainder of the EPF$ routines. If an error status is returned to the caller, epf_id is undefined.

Error Processing

If an error occurs while attempting to allocate dynamic memory space for the EPF or if the user's command environment becomes corrupted, an informative error message will be displayed at the users's terminal and the user's command environment will be reinitialized.

If an error occurs during some manipulation of the in-memory list of EPFs (circular list for instance), an error message is displayed and the user's command environment is reinitialized.

The following error codes may be returned to the caller of EPF$MAP:

E$NMVS	Insufficient VMFA segments available for EPF mapping. The user must either wait until more VMFA segments are returned to the free pool, (by this user or by others), or request that the system be re-configured to allow the user more such segments.
E$NMTS	Insufficent user segments for copying EPF to memory from a remote node or using the K$COPY key.

In response to either of these messages, the user may release temporary segments in these ways:

1. reentering a suspended subsystem via the REENTER command;
2. deactivating previous EPF invocations via the REMEPF command;
3. releasing command levels via the RELEASE_LEVEL command;
4. reinitializing the command environment via the ICE command (as a last resort).

E$ROOM Insufficient dynamic storage is available.

The recommended user action is the same as for E$NMTS above.

E$NRIT User has insufficient access rights to the EPF file.

E$BKEY Invalid key value was specified for EPF$MAP.

E$BUNT The specified unit number is invalid.

E$UNOP File no longer open on specified file unit.

E$NDAM EPF file is not a DAM file, as it must be.

E$NOVA EPF file is not open for VMFA-read, as it must be.

E$FIUS EPF file is currently open for use.

The EPF file may not be mapped probably because it is currently open on a file unit for writing by this or another user.

E$BDAM EPF DAM file structure has been corrupted.

E$IVWN EPF file contents have been corrupted.

E$EPFT Invalid EPF type was detected.

Resubmit the file to BIND.

E$BVER Invalid EPF version was detected.

Resubmit the file to BIND.

E$EPFL EPF too large to be mapped to memory.

EPF$MAP will return this error if the EPF consists of more than 130 procedure segments.

▶ EPF$RUN
(or EPF$RN for FTN calls)

Purpose

This routine performs all the appropriate calls to execute an EPF file. It maps and allocates the linkage and static data areas, initializes them, invokes the EPF, and optionally returns the EPF memory resources to the system free pool. The EPF file must first be opened for a VMFA-read; that is, you first must call either SRCH$$, SRSFX$, or TSR$$ with the K$VMR key specified.

Program EPFs written as programs (that is, they expect no command arguments and return no severity code) are normally invoked with the first calling format below. Those Program EPFs written as functions, and those expecting arguments, must be invoked using the second format below. The additional arguments are provided for passing invocation information to the program being invoked, and for returning data to the invoking program.

Usage

```
dcl epf$run entry (fixed bin (15), fixed bin (15), fixed bin (15))
            returns (fixed bin (31));

epf_id = epf$run (key, unit, status)

                                        or

dcl epf$run entry (fixed bin(15), fixed bin(15),
                   fixed bin(15), char(1024) var, fixed bin(15),
                   1, 2 char(32) var,
                      2 fixed bin(15),
                      2 ptr,
                      2, 3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(11),
                         3 bit(1),
                         3 bit(15),
                         3 fixed bin(15),
                         3 fixed bin(15),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(13),
                   1, 2 bit(1),
                      2 bit(15),
                   ptr);

epf_id = epf$run (key, unit, status, com_args, com_status,
                       com_state, flags, rtn_function_ptr);
```

key (INPUT) Is an action specifier key. Possible values are:

K$INVK Map, create, allocate and initialize static data areas, and store EPF in cache memory upon completion.

K$INVK_DEL (K$IVD for FTN callers) map, allocate and initialize static data areas, invoke and do not cache EPF after completion.

K$REST Map, allocate and initialize static data areas, but do not invoke the EPF.

unit (INPUT) Is the file unit on which the EPF is open for VMFA read, (K$VMR).

status (OUTPUT) Is a standard success/failure code for the invocation of the EPF. Possible values include all error codes returned by EPFMAP, EPFALLC, EPF$INIT, or EPF$DEL.

com_args (INPUT) The command arguments.

com_status (OUTPUT) The command execution error status.

com_state (INPUT) Contains information relative to the EPF invocation. Subdivisions of that information are as follows:

```
1 com_state,
  2 com_name,               /* char(32) var
  2 version,                /* fixed bin(15)
  2 vcb_ptr,                /* ptr
  2 cp_iter_info,
    3 mod_after_date,       /* fixed bin(15)
    3 mod_before_date,      /* fixed bin(15)
    3 bk_after_date,        /* fixed bin(15)
    3 bk_before_date,       /* fixed bin(15)
    3 type_dir,             /* bit(1)
    3 type_segd,            /* bit(1)
    3 type_file,            /* bit(1)
    3 type_acat,            /* bit(1)
    3 type_rbf,             /* bit(1)
    3 res1,                 /* bit(11)
    3 verify_sw,            /* bit(1)
    3 botup_sw,             /* bit(1)
    3 res2,                 /* bit(14)
    3 walk_from,            /* fixed bin(15)
    3 walk_to,              /* fixed bin(15)
    3 in_iteration,         /* bit(1)
```

```
3 in_wildcard,          /* bit(1)
3 in_treewalk,          /* bit(1)
3 res3;                 /* bit(13)
```

with fields that contain the following assignments:

com_name — Name of the EPF command.

version — Version of the com_state structure, set to either 0 or 1; 0 signals that only these first two fields have defined values, while 1 signals that all four of these are defined and provided by the caller.

vcb_ptr — Ptr to local CPL variables allocated during the execution of a CPL program. The field is null () if there is no invoking CPL program involved.

cp_iter_info — Information on the extended command processing features for the command; it is passed to the EPF program from the routine that invoked it.

flags (INPUT) — This field contains information relative to the command function invocation. It has this format:

```
1 flags,
  2 command_function_call bit(1),
  2 reserved bit(15);
```

The first bit, if set, indicates that the program was called as a command function; the remaining fifteen bits are undefined.

rtn_function_ptr (OUTPUT) — Pointer to a return structure for such a function. The memory space for this data will have been allocated by the EPF function. The invoker of the function utilizes this data and later de-allocates the memory space by calling FRE$RA.

epf_id (OUTPUT) — The identifier for the EPF created by a call to EPF$MAP. If the EPF is deleted after its invocation completes, the epf_id may be undefined.

▶ EX$CLR

Purpose

This routine disables the signalling of the EXIT$ condition either after a program's completion or after its termination as the result of a non-local-goto having been executed.

However, to actually disable the EXIT$ condition, one call to EX$CLR must be made for every call to EX$SET, because PRIMOS looks to a single counter that is either incremented or decremented by calls to these two routines.

Usage

```
dcl ex$clr entry ();

call ex$clr;
```

▶ EX$RD

Purpose

This routine returns the state of the counter used to control the conditional signalling of the EXIT$ condition whenever a program EPF terminates. The routine EX$SET enables the EXIT$ condition; the routine EX$CLR disables it.

Usage

```
dcl ex$rd entry (fixed bin(15));

call ex$rd (transmit_exit_setting);
```

transmit_exit_setting
(OUTPUT)

The value returned from the counter, either greater than zero or otherwise. A value greater than zero enables the signalling of the EXIT$ condition whenever a program terminates. If the value is zero or negative, the signal is disabled.

EX$SET

Purpose

This routine enables the signalling of the EXIT$ condition either after a program's completion or after its termination as the result of a non-local goto having been executed.

Usage

```
dcl ex$set entry ();

call ex$set;
```

FRE$RA

Purpose

This routine de-allocates the space designated for EPF functions' return information. After processing the information returned from functions, the invoker should then call this routine to free up space and maintain an efficient command environment.

Usage

```
dcl fre$ra entry (ptr);

call fre$ra (rtn_function_ptr);
```

rtn_function_ptr (INPUT) Pointer to the space set aside for EPF functions, earlier allocated by ALC$RA or ALS$RA.

ICE$

Purpose

This routine initializes the command environment.

It does so by closing all open files, closing the comoutput file, and resetting the command environment. The subroutine never returns, and the invoking program is terminated. A user working in a subdirectory during an ICE$ is therefore returned to the origin UFD.

Usage

```
dcl ice$ entry;

call ice$;
```

> **Caution**
>
> Avoid using this subroutine! It may affect the integrity of subsystems, including Prime data management products. CLEANUP$ on-units on the stack are not invoked. Consequently, it should be used only when the stack has clearly been damaged.

▶ RD$CE_DP
(or RD$CED for FTN calls)

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the current user's profile. This routine returns to the caller the current value of the command environment breadth.

Usage

```
dcl rd$ce_dp entry (fixed bin);

call rd$ce_dp(command_environment_breadth);
```

command_environment_breadth (OUTPUT)	The current breadth of the command environment at this command level.

▶ RPL$

Purpose

This subroutine allows the replacement of one EPF file with another one. By definition, therefore, the file to be replaced must be a DAM file with the suffix .RUN. If the file to be replaced is currently in use (such as an EPF library being accessed by users), it remains in use but has its suffix changed from .RUN to .RPn, where n is a decimal integer from 0 to 9. RPL$ still replaces the old EPF file with this new .RUN file, but the .RPn file continues to exist. Users who now try to access the EPF file are linked to the new .RUN file, but the .RPn continues to exist. Users may later delete or save the old version.

Usage

```
dcl rpl$ entry (char(128) var, char(128) var, char(128) var,
               bit(1) aligned, fixed bin(15));

call rpl$(source_path, target_path, rpl_path, no_query, code);
```

source_path (INPUT) — The file containing the code to be used in the new .RUN file.

target_path (INPUT) — The name of the new .RUN file

rpl_path (OUTPUT) — The name of the old .RUN file, which is now a .RPn file if it is currently in use; otherwise, a null string.

no_query (INPUT) — If this bit is set, no query for changing the file name will prompt the user, and no messages are displayed. If it is unspecified by the user, the routine defaults to query displays.

code (OUTPUT) — The error code. A zero is returned if the subroutine is successful. A -1 is returned as a warning if at least one replace file exists and is not in use, but the user decides not to delete it; other standard subroutine errors (see Appendix D), in the form of E$xxx, also may be returned.

▶ STR$AL

Purpose

This routine allocates space from Dynamic Memory for user-class storage. Instead of raising a success/failure condition (as STR$AU), it returns an informative error code if a problem occurs.

Usage

```
dcl str$al entry(fixed bin(15), fixed bin(31), fixed bin(15),
                 fixed bin (15)) returns(ptr) options(short);

block_ptr =  str$al (reserved, block_size, reserved, status);
```

reserved (INPUT) — This field must have input values of zero.

block_size (INPUT) — The size of the block to allocate.

reserved (INPUT) — This field must contain the value of zero.

status (OUTPUT) — Returned error status. Possible error codes may be:

E$ALSZ	Invalid block-size
E$ROOM	Insufficient space
E$HPER	Corrupt heap

block_ptr — The pointer to the allocated space.

▶ STR$AP

Purpose

This routine allocates space from process-class storage. If any errors are detected, an appropriate error message is displayed and the user's command environment is reinitialized.

Usage

```
dcl str$ap entry(fixed bin(31)) returns(ptr) options(short);

block_ptr = str$ap(block_size);
```

block_size (INPUT) — The size of the block to allocate.

block_ptr — Points to the allocated space.

▶ STR$AS

Purpose

This routine allocates space from dynamic memory for subsystem-class storage. If any errors are detected, an appropriate error code is returned.

Note

> Use STR$AS to allocate dynamic memory space for Prime-supplied subsystems only.

Usage

```
dcl str$as entry(fixed bin(31), fixed bin(15))
        returns(ptr) options(short);

block_ptr = str$as(block_size, code);
```

block_size (INPUT)	The size of the block to allocate.
code (OUTPUT)	Returned error status. Possible error codes may be: E$BPAR Invalid block-size E$ROOM Insufficient space E$NSUC Corrupt heap
block_ptr	The pointer to the allocated space.

▶ STR$AU

Purpose

This routine allocates space from dynamic memory for user-class storage. When a bad block_size is given, it raises the ERROR condition. When not enough space can be found in the heap, it raises the STORAGE condition. When the heap is found to be corrupted, it raises the HEAP_ERROR$ condition.

Usage

```
dcl str$au entry(fixed bin(31)) returns(ptr) options(short);

block_ptr = str$au(block_size);
```

block_size (INPUT)	Size of the block to allocate.
block_ptr	Pointer to the allocated space.

▶ STR$FP

Purpose

This routine returns space to process-class storage. If any errors are detected, an appropriate error message is displayed and the user's command environment is reinitialized.

Usage

```
dcl str$fp entry(ptr) options(short);

call str$fp (block_ptr);
```

block_ptr (INPUT)	Pointer to the allocated space.

▶ STR$FR

Purpose

This routine returns space to user-class storage. If any errors are detected, an error code is returned (instead of an error condition as with STR$FU).

Usage

```
dcl str$fr entry(fixed bin(15), ptr, fixed bin(15));

call str$fr (reserved, block_ptr, status);
```

reserved (INPUT) — Reserved.

block_ptr (INPUT) — The pointer to the storage space to be freed.

status (OUTPUT) — The returned error status. Possible error codes may be:

E$FRER	Invalid free request
E$HPER	Corrupted heap

▶ STR$FS

Purpose

This routine returns space to subsystem-class storage. If any errors are detected, an appropriate error code is returned.

Usage

```
dcl str$fs entry(ptr, fixed bin(15));

call str$fs(block_ptr, code);
```

block_ptr (INPUT) — The pointer to the allocated space.

code (OUTPUT) — Returned error status. Possible error codes may be:

E$FRER	Invalid free request
E$NSUC	Corrupted heap

▶ STR$FU

Purpose

This routine returns space to user-class storage. When a bad block_ptr is passed, it raises the ERROR condition. When the heap is found to be corrupted, it raises the HEAP_ERROR$ condition.

Usage

```
dcl str$fu entry(ptr);

call str$fu(block_ptr);
```

block_ptr (INPUT)	Pointer to block of data to free

▶ ST$SGS

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the current user's profile. This routine retrieves the maximum number of private, static segments allocated to the user.

Usage

```
dcl st$sgs entry () returns (fixed bin(15));

maximum_private_static_segs = st$sgs();
```

M
Other New Subroutines at Revision 19.4

The following subroutine descriptions are also released for Revision 19.4.

Subroutine	Function
COMLV$	Call a new command level.
CMLV$E	Call a new command level upon an error condition.
EQUAL$	Generate a new name for an established object name.
LIST$CMD	Display those mini-level commands qualified by a wildcard character string match.
LON$CN	Enable or disable logout notification for phantoms.
LV$GET	Retrieve the value of a local variable defined within a CPL program.
LV$SET	Set the value of a local variable within a CPL program.
RSEGAC$	Identify a user's access rights to a particular segment.

TTY$RS Clear the current user's input and output buffers.

Note

The following subroutine descriptions use a PL/I-like format to supply a base for consistency in the presentation of data structures.

▶ COMLV$

Purpose

This routine causes a new command level to be called. A PRIMOS routine called the command listener is indirectly invoked, displays the OK prompt message, and waits for input. Only after the user issues the START command from that command level will the COMLV$ subroutine return to the caller. Use this routine under normal conditions (not error conditions, which require cmlv$e).

Usage

```
dcl comlv$ entry ();

call comlv$;
```

▶ CMLV$E

Purpose

This routine causes a new command level to be called upon an error condition. A PRIMOS routine call the command listener, indirectly invoked, does the following: it pauses command input; it displays the ER prompt message; it waits for input; it forces terminal output on; it enables quits. Only after the user issues a START command from that command level will the CMLV$E subroutine return to the caller.

Usage

```
dcl cmlv$e entry();

call cmlv$e;
```

▶ EQUAL$

Purpose

This routine expects an object name and a generation pattern. The latter contains "commands" that specify how to transform the object name into a new name called the generated name. This routine performs that transformation. Name generation is discussed in the PRIMOS Commands Reference Guide.

Usage

```
dcl equal$ entry (char(32) var, char(32) var, char(32) var,
                  fixed bin(15));

call equal$ (obj_name, pattern, generated, code);
```

obj_name (input) — The object name being submitted for transformation into the new name.

pattern (input) — A character string that contains the generation pattern of commands to carry out the transformation.

generated (output) — The new object name generated according to pattern.

code (output) — The standard error code — see Appendix D (zero indicates success).

▶ LIST$CMD

Purpose

This routine displays to a user's terminal those mini-level commands qualified by a wildcard character string match. The command mini-level is explained in the Programmer's Guide to BIND and EPFs.

Usage

```
dcl list$cmd entry (char(32) var, fixed bin(15));

call list$cmd (wildcard_match, status);
```

wildcard_match (INPUT/OUTPUT)
The wildcard character string submitted for a search and match. Any matches found are returned herein.

status (OUTPUT)
Any error code to be returned to the caller of the routine. If the wildcard string submitted is invalid, an error code such as E$FDMM (format/data mismatch) is returned. If a valid string does not elicit a single match, E$FNTF (file not found) is returned.

▶ LON$CN

Purpose

This routine performs logout notification for phantoms, if passed a proper value within the key. If it receives an improper value, it simply ignores the call.

Usage

```
dcl lon$cn entry (fixed bin (15));

call lon$cn (key);
```

key (INPUT)
Any values other than the following are ignored:

0 Turn notification off.
1 Turn notification on.

▶ LV$GET

Purpose

An EPF command invoked from a CPL program uses this routine to retrieve the value of a variable defined within that CPL program.

Usage

```
dcl lv$get entry (ptr, char(32) var, char(*) var,
                  fixed bin(15), fixed bin(15));

call lv$get (vcb_ptr, var_name, var_value, var_size, code);
```

vcb_ptr (INPUT) — The pointer to the block of local variables for the CPL program.

var_name (INPUT) — The name of the local variable in the CPL program.

var_value (OUTPUT) — The value of the CPL local variable.

var_size (Output) — The maximum length in characters of the user buffer, var_value.

code (OUTPUT) — The standard return error code from Appendix D.

▶ LV$SET

Purpose

An EPF command invoked from a CPL program uses this routine to set the value of a local variable within the CPL program.

Usage

```
dcl lv$set entry (ptr, char(32) var,
                  char(*) var, fixed bin(15));

call lv$set (vcb_ptr, var_name, var_value, code);
```

vcb_ptr (INPUT) — The pointer to the block of local variables for the CPL program.

var_name (INPUT) — The name of the local variable in the CPL program.

var_value (Input) — The value of the CPL local variable.

code (OUTPUT) — The standard return error code from Appendix D.

▶ RSEGAC$

Purpose

This routine is used to verify that a particular segment exists. It also indicates the requester's access rights to the segment. If the segment does not exist, the "if rsegac$" call elicits a return FALSE ('0'). If the segment exists, a TRUE ('1') is returned and the access value for that segment is also returned in the access argument.

FORTRAN programs cannot directly call this subroutine, because it has a seven-character name. A given program may indirectly call it, for example, with "call synym(segno, access)", and at BIND time rename synym as rsegac$.

Usage

```
dcl rsegac$ entry (fixed bin(15), fixed bin(31))
          returns (bit(1));

if rsegac$ (segno, access)
   then.......;
```

segno (INPUT) The segment number in question.

access (OUTPUT)

1. The first halfword is reserved.
2. If the segment exists, the value returned indicates the user's access rights to the segment. Possible values and their interpretations are:

0	No access
1	Gate
2	Read Access
3	Read, Write Access
4,5	Reserved
6	Read, Execute Access
7	Read, Write, Execute Access

▶ TTY$RS

Purpose

This routine is called by the QUIT$ handler to clear the current user's input and output buffers. A key is passed that contains two bits specifying whether the input and output buffers are to be cleared. This routine takes no action for non-interactive users (such as phantoms and batch jobs).

Usage

```
dcl tty$rs entry (fixed bin(15), fixed bin(15));

call tty$rs (key, code);
```

key (INPUT) — The keys indicating whether or not to clear the I/O buffers. Possible key values are:

K$OUTB	Clear output buffer
K$INB	Clear input buffer

code (OUTPUT) — The standard error return code from Appendix D.

N
The C Interface

INTRODUCTION

A subroutine can be called from C by calling the subroutine's name and the arguments to be used in the program. For example:

```
    sub-name [(argument 1...argument n)];
```

DATA TYPES

Table N-1 summarizes the data types of FORTRAN and PL1G subroutines and functions that can be called from C. The section that follows describes the C data types and their FORTRAN and PL1G equivalents.

Table N-1
Data Types

GENERIC UNIT/PMA	BASIC/ VM	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL1G	C

1 bit	-*-	-*-	-*-	-*-	-*-	(1)
						Bit
						Bit(1)
							struct{
							unsigned;
							}name;

16-bit Halfword	INT	COMP	(2) INTEGER INTEGER*2 LOGICAL	(2) INTEGER*2 LOGICAL*2	(3) Integer Boolean	Fixed Bin Fixed Bin(15)	Short Int
32-bit Word	INT*4	-*-	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	(4) Subrange	Fixed Bin(31)	(Long) Int
64-bit Double Word	-*-	-*-	-*-	-*-	-*-	-*-	-*-
32-bit Float single precision	REAL	-*-	REAL REAL*4	REAL REAL*4	Real	Float Binary Float Bin (23)	Short Float
64-bit Float double precision	REAL*8	-*-	REAL*8	REAL*8	-*-	Float Bin(47)	Long Float
Byte string (Max. 32767)	INT	DISPLAY(5) PIC A(n) PIC 9(n) PIC X(n)	INTEGER	(5) CHARACTER *n	(5) ARRAY [1..n] OF CHAR	(5) Char(n) Name[ ]	Char
Varying (6) character string	-*-	(6)	(6)	(6)	(6)	(6) Char(n) Varying	
(7) 48-bits 3 Halfwords	-*-	-*-	-*-	-*-	(8) ^<type>	Pointer	Pointer

* Not available.

Notes to Table N-1

(1) If used for representing true (1) and false (0), negative numbers are true, positive numbers and 0 are false. This is not compatible with FORTRAN. In PL1G, '1'B is true; if this value is stored in a 16-bit integer, the sign bit is set, giving 100000 octal, or -32768 decimal. False in PL1G may always be represented as decimal 0.

(2) LOGICAL data in FORTRAN represents true and false as 1 and 0, respectively. This is not directly compatible with Pascal or PL1G.

(3) Boolean data in Pascal is represented in 16 bits where the sign bit determines true and false. (A negative sign means true, a positive sign means false.) This data type is directly compatible with a BIT(1) ALIGNED variable in PL1G.

(4) To define a 32-bit integer in Pascal, use an integer array whose positive limit is greater than 32768 and whose negative limit is less than -32768.

(5) Where "n" is a constant expression with the program module. This is not a dynamic length.

(6) A character-varying string can be simulated in each language indicated, as discussed in the chapter on that language.

(7) This implementation of a pointer in PL1G is subject to change; a program that passes pointers or receives them may have to be recompiled, and a program that assumes a particular form or size of pointer data may have to be rewritten.

(8) Where <type> is either a user-defined type or a standard Pascal type.

INTEGER*2 or FIXED BIN

The INTEGER*2 data type expected by FORTRAN subroutines is the FIXED BIN or FIXED BIN (15) data type in PL1. These two data types must be declared as SHORT INT in C programs.

Sample program 1 illustrates a call to the FORTRAN subroutine SRCH$$ that expects an INTEGER*2 data type.

INTEGER*4 or FIXED BIN(31)

The INTEGER*4 data type expected by FORTRAN subroutines is the FIXED BIN(31) data type in PL1G. The C equivalent for these two data types is LONG INT or simply INT. Sample program 2 calls the subroutine RNUM$A that expects an INTEGER*4 data type.

REAL*4 or FLOAT BIN(23)

REAL*4 or FLOAT BIN(23) data types expected by FORTRAN and PL1G subroutines respectively should be declared as FLOAT in C programs. Sample program 3 calls the FORTRAN subroutine RAND$A that expects a REAL*4 data type.

REAL*8 or FLOAT BIN(47)

The REAL*8 data type expected by FORTRAN subroutines is the FLOAT BIN(47) data type in PL1G. These two data types must be declared as DOUBLE in C.

LOGICAL*1

This FORTRAN data type must be declared as CHAR in C programs, with only the low order bit of the character being used. Sample program 4 calls the FORTRAN subroutine DELE$A that expects a LOGICAL*1 data type.

Pointers

A POINTER data type expected by a PL1G subroutine can also be declared as a POINTER data type in C programs. Note that the use of any other C data type to receive a pointer argument may cause unpredictable results. Sample program 5 calls a PL1G subroutine that expects a POINTER data type.

Enumeration Data Type

This C data type is analogous to the scalar data type in PASCAL. Enumeration data types are defined by declaring a type specifier followed by an ordered list of identifiers, which are declared as constants. The enumeration type specifier and the identifiers used must all be unique. All enumeration identifiers are assumed to be of the data type INT. There is no equivalent data type in FORTRAN and PL1G.

Void Data Type

This C data type implies a nonexistent value, which cannot be used in any way. Expressions derived from this data type can only be used as an expression statement or as the left operand of a comma expression. An expression can be converted to a data type of void by use of the cast operator. There is no equivalent data type in FORTRAN and PL1G.

Integer Arrays and Character Arrays

Arrays expected by FORTRAN and PL1G subroutines should be declared as an array of integers or as an array of characters in C, depending on the type of array being passed. However, a FORTRAN integer array can contain both integer and character data, which must be declared differently in C. In this case, the C argument must be declared as a structure containing both data types.

Sample program 6 calls the PRIMOS subroutine TIMDAT to retrieve user and system information. Note that the integer array returned by TIMDAT contains both integer and character data.

ASCII Character Strings

An ASCII character string expected by a FORTRAN or PL1G subroutine should be declared as a string literal or character array in C, as in Sample Program 2.

CHARACTER*VARYING

This PL1G data type is implemented as a record structure, providing a count of the number of characters in the structure followed by the actual characters themselves. The structure of a CHAR(*)VAR argument may be represented as follows:

05	A B C D E
COUNT	CHARACTER STRING

Sample program 7 uses a CHAR(*)VAR data type.

THE -NOCONVERT OPTION

If a C subroutine is being called from another Prime-supported language (for example, FORTRAN or PL1G), the conversion of char, short, and float data types does not occur. The C compiler, however, is not aware of this. Therefore, the -NOCONVERT compiler option must be used to inform the C compiler that data types of char, short, and float should not be converted. See the C User's Guide for more information on data type conversion and the -NOCONVERT compiler option.

THE FORTRAN STORAGE CLASS

If a C function is calling a subroutine from another Prime- supported language that expects data types of char, short, or float, then the implicit C default action of converting these data types must be prevented. The FORTRAN storage class can be used to prevent char and short data types from being converted to long, and the float data type from being converted to double.

When used with the ampersand operator (&), the FORTRAN keyword disables default data type conversion. As a result of this, the data type is passed by reference rather than by value. The examples in this chapter all use the fortran storage class for PRIMOS subroutines.

MORE ABOUT C

Additional information on accessing common blocks, creating common blocks from C, transferring arguments in C, and passing arrays by reference can be found in the C User's Guide.

USING SYSCOM TABLES

Subroutine descriptions in this guide ocassionally refer to codes having names in the format x$yyyy, where x and y are letters. These codes can be substituted for numeric values and should be inserted at the beginning of a C module. There are three groups of these codes for C.

Error codes have names in the format E$yyyy. These equivalents should be inserted in the C program with the statement:

```
    #include <errd.ins.cc>
```

Key codes have names in the format K$yyyy. These key codes should be inserted in the C program with the statement:

```
    #include <keys.ins.cc>
```

Subroutines in the VAPPLB use argument codes in the form A$yyyy. The numeric equivalents of these codes are in the file SYSCOM>A$KEYS. Sample programs 1, 2, and 3 illustrate the use of SYSCOM tables. At Revision 19.4, you must declare the numeric equivalent in the C program, as is done for A$DEC in sample program 2. The BIND subcommand LI VAPPLB must also be issued at load time.

SAMPLE PROGRAMS

The remainder of the chapter will provide a number of sample programs illustrating the use of error codes, key codes, and the various data types described above.

Program 1 — Using an Integer*2 Argument

This program calls the FORTRAN subroutine SRCH$$ that expects a data type of INTEGER*2. This sample program also illustrates use of SYSCOM tables.

```
    OK, SLIST SRCH.CC

    /*Calls the subroutine SRCH$$ to check for the existence of a
      file*/

    #include <keys.ins.cc>
    #include <errd.ins.cc>
```

```
main()
{
  short key, name_len, funit, type, code;
  fortran srch$$();

  key = k$exst + k$iufd;
  name_len = 6;
  funit = 0;
  type = 0;

  srch$$ (&key, "ctrlfl", &name_len, &funit, &type, &code);
  printf ("returned error code is %d\n", code);
}
```

This program can be compiled, loaded, and executed in V-mode with the following dialog. If the file "ctrlfl" is not found,

```
OK, CC SRCH
[CC revision 19.4]
00 Errors and 00 Warnings detected in 17 source lines and 288
include lines.

OK, BIND SRCH
[BIND rev 19.4]
$ LO SRCH.BIN
$ LI C_LIB
$ LI
BIND COMPLETE
$FILE

OK, R SRCH.RUN
returned error code is 15
```

Program 2 — Using an INTEGER*4 Argument

```
/*Calls the FORTRAN subroutine RNUM$A to verify a data type of
  INTEGER*4*/
```

```
OK, SLIST RNUM.CC

main()
{
   static char msg[21] = "please enter a number:";
   short msglen, a$dec;
   int value;
   fortran rnum$a();
```

```
      msglen = 21;
      a$dec = 1;
      rnum$a (msg, &msglen, &a$dec, &value);
      printf ("the number is %d\n", value);
   }
```

This program can be compiled, loaded, and executed in V-mode with the following dialog:

```
OK, CC RNUM
[CC revision 19.4]
00 Errors and 00 Warnings detected in 13 source lines.


OK, BIND RNUM
[BIND rev 19.4]
$ LO RNUM.BIN
$ LI C_LIB
$ LI VAPPLB       /*Requires use of V-mode application library*/
BIND COMPLETE
$ FILE

OK, R RNUM.RUN
please enter a number:  3685
the number is 3685
```

Program 3 — Using a REAL*4 Argument

/*Calls the FORTRAN subroutine RAND$A to generate random numbers*/

```
OK, SLIST RAND.CC

main()
{
   int seed;
   float number;
   short k;
   fortran float rand$a();

   seed = 1;
   for (k=1; k<=10; k++)
     {
       number = rand$a (&seed);
       printf ("%e\n", number);
     }
}
```

This program can be compiled, loaded, and executed in V-mode with with the following dialog:

```
OK, CC RAND
[CC revision 19.4]
00 Errors and 00 Warnings detected in 14 source lines.


OK, BIND RAND
[BIND rev 19.4]
$ LO RAND.BIN
$ LI C_LIB
$ LI VAPPLB     /*Requires use of V-mode application library*/
BIND COMPLETE
$ FILE

OK, R RAND.RUN
7.826369e-6
1.315378e-1
7.556052e-1
4.586501e-1
5.327672e-1
2.189592e-1
4.704461e-2
6.788646e-1
6.792964e-1
9.346928e-1
```

Program 4 — Using a LOGICAL*1 Argument

/*Calls the FORTRAN subroutine DELE$A to delete a file*/

```
OK, SLIST DELE.CC

main()
{
   static char filename[7] = "ctrlfl";
   short count = 6;
   fortran short dele$a();
   char log;

   log = dele$a (filename, &count);
   if (log == 1)
      printf ("file deleted successfully\n");

   else

      printf ("no go\n");
}
```

This program can be compiled, loaded, and executed in V-mode with the following dialog:

```
OK, CC DELE
[CC revision 19.4]
00 Errors and 00 Warnings detected in 13 source lines.


OK, BIND DELE
[BIND rev 19.4]
$ LO DELE.BIN
$ LI C_LIB
$ LI VAPPLB     /*Requires use of V-mode application library*/
BIND COMPLETE
$ FILE

OK, R DELE.RUN
file deleted successfully
```

Program 5 — Using a POINTER Argument

```
/*Calls the PL1G subroutine AC$SET to create ACLs for a file.
  An error message is returned if the file contains ACLs.*/


    OK, SLIST ACSET.CC

    main()
    {
       short key, code;
       struct name
             {
              short number;
              char filename[128];
             };
    static struct name name1 = {22, "mine>c>techpub>acltest"};
    struct acl{
               short version;
               short num;
               struct name entries;
              };
    static struct acl ctrlfl = {2, 1, 8, "mine:all"};
    short *acl_ptr;
    fortran ac$set();

    key = 0;
    acl_ptr = &ctrlfl;
    ac$set (&key, &name1, acl_ptr, &code);
    printf ("error code is: %d\n", code);
    }
```

This program can be compiled, loaded, and executed in V-mode with the following dialog:

```
OK, CC ACSET
[CC revision 19.4]
00 Errors and 00 Warnings detected in 24 source lines.


OK, BIND ACSET
[BIND rev 19.4]
$ LO ACSET
$ LI C_LIB
$ LI
BIND COMPLETE
$ FILE

OK, R ACSET.RUN
error code is: 189
```

Program 6 — Using an INTEGER ARRAY Argument

```
/*Calls the PRIMOS subroutine TIMDAT to retrieve system and user
  information*/
```

```
OK, SLIST TIMDAT.CC

main()
{
   static struct array
       {
        char mmddyy[6];
        short time_min;
        short time_sec;
        short time_tck;
        short cpu_sec;
        short cpu_tck;
        short disk_sec;
        short disk_tck;
        short tck_sec;
        short user_num;
        char username[32];
       };

static struct array intarray;
short num = 28;
fortran timdat();
```

```
    timdat(&intarray, &num);
    printf ("date is                   %s\n", intarray.mmddyy);
    printf ("seconds elapsed           %d\n", intarray.time_sec);
    printf ("ticks elapsed             %d\n", intarray.time_tck);
    printf ("cpu seconds used          %d\n", intarray.cpu_sec);
    printf ("cpu ticks                 %d\n", intarray.cpu_tck);
    printf ("disk seconds used         %d\n", intarray.disk_sec);
    printf ("user name                 %32s\n", intarray.username);
    }
```

This program can be compiled, loaded, and executed in V-mode with the following dialog:

```
    OK, CC TIMDAT
    [CC revision 19.4]
    00 Errors and 00 Warnings detected in 31 source lines.


    OK, BIND TIMDAT
    [BIND rev 19.4]
    $ LI CCMAIN
    $ LO TIMDAT.BIN
    $ LI C_LIB
    $ LI
    BIND COMPLETE
    $ FILE

    OK, R TIMDAT.RUN
    date is                 022585
    seconds elapsed         54
    ticks elapsed           78
    cpu seconds used        19
    cpu ticks               190
    disk seconds used       8
    user name               RCJ
```

Program 7 — Using a CHAR*VAR Argument

```
/*Calls the PL1G subroutine GV$GET to obtain the value of the global
  variable .MAX*/
```

```
OK, SLIST CHARVAR.CC

main()
{
  short varsize, code;
  struct charvar
     {
     short nchars;
     char string1[5];
     };
static struct charvar varname = {4, ".max"};
static struct charvar varvalue;
fortran gv$get();

varsize  = 5;
gv$get (&varname, &varvalue, &varsize, &code);
printf ("value of global variable .max is %s\n", varvalue.string1);
printf ("error code is %d\n", code);
}
```

This program can be compiled, loaded, and executed in V-mode with the following dialog, providing that a global variable file has been previously established as explained in the CPL User's Guide.

```
OK, CC CHARVAR
[CC revision 19.4]
00 Errors and 00 Warnings detected in 17 source lines.


OK, BIND CHARVAR
[BIND rev 19.4]
$ LO CHARVAR.BIN
$ LI C_LIB
$ LI
BIND COMPLETE
$ FILE

OK, R CHARVAR.RUN
value of global variable .max is 100
error code is 0
```

O Corrections

Please make the following additions and revisions to the Subroutines Reference Guide.

On page 9-17 for the subroutine GPATH$:

for key	Delete the numerical references: (K$UNIT=1), (K$CURA=2), (K$HOMA=3).

On page 9-29 for the subroutine RDEN$$:

for buflen	Add after (INTEGER*2) "set to a value of 24".

On page 9-48 for the subroutine SRCH$$:

under action	Add the following key option: K$VMR Open filname for VMFA read.

On page 9-57 for the subroutine SRSFX$:

for status	Replace the description with: The standard status returned, either a 0 to signal a success or an error code from those listed in Appendix D.

On page 9-58 for the subroutine TSRC$$:

under action	Add the following key option: K$VMR Open pathname for VMFA read.

On page 10-2 within Table 10-1 Operating System Subroutines:

under Read or Write change C1IN$ to C1IN.

On page 10-3 for the subroutine C1IN$:

Change the title and its call from C1IN$ to C1IN. (This correction was already given the 19.3 MRU but is repeated here for the reader's convenience.)

On page 10-4 for the subroutine CL$GET:

under Purpose	Delete from the last sentence: ..or one consisting of all blanks....

On page 10-11 within the discussion of the subroutine CL$PIX delete the last specified data type:

file	Primos filename.

Also, on page 10-12 delete the last entry from the data type correspondence table:

file	char(128) var	INTEGER(65).

On page 10-43 for the subroutine RECYCL:

under Purpose	Add the following sentences at the end: This subroutine is obsolete. To create a controlled delay, use SLEEP$.

On page 12-21 for the subroutine CTIM$A

under cputim — Delete "character string format."

under Discussion — Delete "...either REAL*4 or" (REAL*8 is correct).

On page 19-9 for the subroutine SPOOL$:

under key — Insert these additional user options:

3 Modify the attributes of a file already spooled.

4 Close file on unit info(2) in queue and notify semaphore.

for info — Replace its entire description, as well as all the ensuing descriptions for SPOOL$ with the following information. The lines involving changes or additions are marked with revision bars.

info — Information array of 40 16-bit halfword elements, as follows:

1 Reserved after Rev. 17.

2 Open print file on this unit (key=2). If unit number is zero, then SPOOL$ will return the unit number here.

3 Print option element. (See bit descriptions for element 3 below).

4-6 Form type (6 ASCII characters). (Equivalent to -FORM on PRIMOS command line.)

7 Plot raster scan size (plot only). This represents the number of halfword/raster scan.

8-10 Spool filename (returned).

11 Deferred print time (valid only if defer bit (#8) is set in element 3) — an integer specifying minutes after midnight (equivalent to -DEFER in PRIMOS command line).

12 File size, returned if key is 1.

13-20 (Optional) Logical destination name — must be blank padded (equivalent to -AT on PRIMOS command line). If these elements are used, bit 10 of element 3 (print option element) must be set to 1.

21-28 (Optional) Substitute filename to be used — must be blank padded (equivalent to -AS on command line.) If these elements are used, bit 11 of info(3) must be set to 1.

29 (Optional) Number of copies (equivalent to -COPIES on command line). If this element is used, bit 12 of info(3) must be set.

The remaining 11 elements are for the extended array. If the extended array is used, bit 16 of info(3) must be set.

30 Extended print option element. (See bit descriptions for the extended print option element (info(30)) below.)

31 Disk number of the SPOOLQ satisfying the -DISK option. If this element is used, bit 1 of info(30) must be set.

32-40 Reserved.

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 16-bit halfwords long. Copy time is inversely proportional to buffer size. Nominal size is between 300 and 2000 halfwords.

buflen Length of buffer in halfwords.

code Return code (nonzero if error occurred).

The print option element (info(3)) specifies various print(/plot) information and is defined as follows:

Bit	Designates, if Set
1	Fortran format control. (Column 1 contains carriage control information.)
2	Expand compressed listing.
3	Generate line numbers 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 (not used after Rev. 17).
10	If set, use the logical destination name specified in info(13-20).
11	If set, use the substitute filename specified in info(21-28).
12	If set, spool the number of copies specified in info(29).
13-14	Reserved.
15	Inform user when file has completed printing.
16	Extended array used — MUST be set if info(30-40) used.

The extended print option element (info(30)) specifies additional information and is defined as follows:

Bit	Designates, if Set
1	Attach to the SPOOLQ on disk number in info(31).
2	This file is a PRIORITY file. It can only be used in conjunction with the -MODIFY key.
3	Used in conjunction with -MODIFY to remove the PRIORITY attribute from this file.
4	Allow SPOOL$ to place a message in name when code is not 0 (the usual input to ERRPR$ using the spool command).
5-6	Reserved.
7	Truncate all lines to the value defined by PROP WIDTH command.
8	Used in conjunction with -MODIFY to remove the DEFER attribute from a file.
9-16	Reserved.

On page 22-22 for the subroutine RVONU$:

within DCL line	Change "CHAR(*)" to "CHAR(32)".

On page 22-47ff change the format of the stack-frame header as indicated by the revision bars:

```
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 stk_cbits bit(1),
              3 lib_proc bit(1),
              3 ecb_cbits bit(1),
              3 mbz bit(6),
              3 fault_fr bit(2),
                 2 root,
                   3 mbz bit(4),
                   3 seq_no bit(12),
                 2 ret_pb ptr options (short),
                 2 ret_sb ptr options (short),
                 2 ret_lb ptr options (short),
                 2 ret_keys bit(16) aligned,
                 2 after_pcl fixed bin,
                 2 hdr_reserve(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),
                 2 reserved(6), fixed bin,
                 2 cond_bits bit(16) aligned;


dcl       1 ecb based, /* Entry Control Block */
                 2 pb ptr options (short),
                 2 frame_size fixed bin(15),
                 2 stack_seg fixed bin(12),
                 2 arg_offset fixed bin(15),
                 2 num_args fixed bin(15),
                 2 lb ptr options (short),
                 2 cond_bits bit(16) aligned,
                 2 reserved(6) fixed bin(15);
```

Add the descriptions of new fields in the stack-frame header to those on page 22-48ff.

After flags.user_proc on page 22-48 insert the following descriptions:

flags.stk_cbits — If '1'b, then cond_bits exists within the stack frame header and should be used to determine whether to signal an exception condition. If '0'b, then flags.ecb_cbits is checked.

flags.lib_proc — If '1', then the procedure is a library routine.

flags.ecb_cbits — If '1', then ecb.cond_bits exists and should be used to determine whether to signal an exception condition. If both flags.stk_cbits and flags.ecb_cbits are '0', then flags.- lib_proc is examined.

Note

If all three of the previous flag bits are reset ('0'), then PL/I default condition handling is used.

After next_efh on page 22-50 insert the following field descriptions:

reserved	Reserved.
cond_bits	PL/I condition enable bits.

On page A-11 for the subroutine AT$:

under Discussion   Delete the third and fourth bulleted items:

- A partition name of "<>" means any partition.
- A bare partition name indicates the MFD.

On page A-26 for the subroutine DIR$LS in the field descriptions for the returned directory entry structure:

dtm   Change "dtm" to "dtb" and replace the first sentence of the description with:
The date/time that the BACKUP utility was last run to save the object.

On page A-30 for the subroutine GETID$:

version   In the second sentence change the version number for Rev. 19 from "2" to "1 or 2".

On page D-2ff replace the error code listing with the following updated version:

```
/* ERRD.INS.PL1, PRIMOS>INSERT, PRIMOS GROUP, 01/29/85
   MNEMONIC CODES FOR FILE SYSTEM (PL1)
   Copyright (c) 1982, Prime Computer, Inc., Natick, MA 01760 */
/********************************************************************/

      /*******************************************************/
      /*                                                     */
      /*                                                     */
      /*       CODE DEFINITIONS                              */
      /*                                                     */
      /*                                                     */
      E$EOF  BY  00001, /* END OF FILE                  PE        */
      E$BOF  BY  00002, /* BEGINNING OF FILE            PG        */
      E$UNOP BY  00003, /* UNIT NOT OPEN                PD,SD     */
      E$UIUS BY  00004, /* UNIT IN USE                  SI        */
      E$FIUS BY  00005, /* FILE IN USE                  SI        */
      E$BPAR BY  00006, /* BAD PARAMETER                SA        */
      E$NATT BY  00007, /* NO UFD ATTACHED              SL,AL     */
      E$FDFL BY  00008, /* UFD FULL                     SK        */
      E$DKFL BY  00009, /* DISK FULL                    DJ        */
      e$disk_full
             by      9, /* alias to E$DKFL                        */
      E$NRIT BY  00010, /* NO RIGHT                     SX        */
      E$FDEL BY  00011, /* FILE OPEN ON DELETE          SD        */
      E$NTUD BY  00012, /* NOT A UFD                    AR        */
      E$NTSD BY  00013, /* NOT A SEGDIR                 —         */
      E$DIRE BY  00014, /* IS A DIRECTORY               —         */
      E$FNTF BY  00015, /* (FILE) NOT FOUND             SH,AH     */
      E$FNTS BY  00016, /* (FILE) NOT FOUND IN SEGDIR   SQ        */
      E$BNAM BY  00017, /* ILLEGAL NAME                 CA        */
      E$EXST BY  00018, /* ALREADY EXISTS               CZ        */
      E$DNTE BY  00019, /* DIRECTORY NOT EMPTY          —         */
      E$SHUT BY  00020, /* BAD SHUTDN (FAM ONLY)        BS        */
      E$DISK BY  00021, /* DISK I/O ERROR               WB        */
      E$BDAM BY  00022, /* BAD DAM FILE (FAM ONLY)      SS        */
      E$PTRM BY  00023, /* PTR MISMATCH (FAM ONLY)      PC,DC,AC */
      e$rec_hdr_ptr_mismatch
             by     23, /* alias to E$PTRM                        */
      E$BPAS BY  00024, /* BAD PASSWORD (FAM ONLY)      AN        */
      E$BCOD BY  00025, /* BAD CODE IN ERRVEC           —         */
      E$BTRN BY  00026, /* BAD TRUNCATE OF SEGDIR       —         */
      E$OLDP BY  00027, /* OLD PARTITION                —         */
      E$BKEY BY  00028, /* BAD KEY                      —         */
      E$BUNT BY  00029, /* BAD UNIT NUMBER              —         */
      E$BSUN BY  00030, /* BAD SEGDIR UNIT              SA        */
      E$SUNO BY  00031, /* SEGDIR UNIT NOT OPEN         —         */
      E$NMLG BY  00032, /* NAME TOO LONG                —         */
      E$SDER BY  00033, /* SEGDIR ERROR                 SQ        */
      E$BUFD BY  00034, /* BAD UFD                      —         */
      E$BFTS BY  00035, /* BUFFER TOO SMALL             —         */
      E$FITB BY  00036, /* FILE TOO BIG                 —         */
```

```
E$NULL BY  00037, /* (NULL MESSAGE)                  —         */
E$IREM BY  00038, /* ILL REMOTE REF                  —         */
E$DVIU BY  00039, /* DEVICE IN USE                   —         */
E$RLDN BY  00040, /* REMOTE LINE DOWN                —         */
E$FUIU BY  00041, /* ALL REMOTE UNITS IN USE         —         */
E$DNS  BY  00042, /* DEVICE NOT STARTED              —         */
E$TMUL BY  00043, /* TOO MANY UFD LEVELS             —         */
E$FBST BY  00044, /* FAM - BAD STARTUP               —         */
E$BSGN BY  00045, /* BAD SEGMENT NUMBER              —         */
E$FIFC BY  00046, /* INVALID FAM FUNCTION CODE       —         */
E$TMRU BY  00047, /* MAX REMOTE USERS EXCEEDED       —         */
E$NASS BY  00048, /* DEVICE NOT ASSIGNED             —         */
E$BFSV BY  00049, /* BAD FAM SVC                     —         */
E$SEMO BY  00050, /* SEM OVERFLOW                    —         */
E$NTIM BY  00051, /* NO TIMER                        —         */
E$FABT BY  00052, /* FAM ABORT                       —         */
E$FONC BY  00053, /* FAM OP NOT COMPLETE             —         */
E$NPHA BY  00054, /* NO PHANTOMS AVAILABLE           -         */
E$ROOM BY  00055, /* NO ROOM                         —         */
E$WTPR BY  00056, /* DISK WRITE-PROTECTED            JF        */
E$ITRE BY  00057, /* ILLEGAL TREENAME                FE        */
E$FAMU BY  00058, /* FAM IN USE                      —         */
E$TMUS BY  00059, /* MAX USERS EXCEEDED              —         */
E$NCOM BY  00060, /* NULL_COMLINE                    —         */
E$NFLT BY  00061, /* NO_FAULT_FR                     —         */
E$STKF BY  00062, /* BAD STACK FORMAT                —         */
E$STKS BY  00063, /* BAD STACK ON SIGNAL             —         */
E$NOON BY  00064, /* NO ON UNIT FOR CONDITION        —         */
E$CRWL BY  00065, /* BAD CRAWLOUT                    —         */
E$CROV BY  00066, /* STACK OVFLO DURING CRAWLOUT     —         */
E$CRUN BY  00067, /* CRAWLOUT UNWIND FAIL            —         */
E$CMND BY  00068, /* BAD COMMAND FORMAT              —         */
E$RCHR BY  00069, /* RESERVED CHARACTER              —         */
E$NEXP BY  00070, /* CANNOT EXIT TO COMMAND PROC     —         */
E$BARG BY  00071, /* BAD COMMAND ARG                 —         */
E$CSOV BY  00072, /* CONC STACK OVERFLOW             —         */
E$NOSG BY  00073, /* SEGMENT DOES NOT EXIST          —         */
E$TRCL BY  00074, /* TRUNCATED COMMAND LINE          —         */
E$NDMC BY  00075, /* NO SMLC DMC CHANNELS            —         */
E$DNAV BY  00076, /* DEVICE NOT AVAILABLE           DPTX       */
E$DATT BY  00077, /* DEVICE NOT ATTACHED             —         */
E$BDAT BY  00078, /* BAD DATA                        —         */
E$BLEN BY  00079, /* BAD LENGTH                      —         */
E$BDEV BY  00080, /* BAD DEVICE NUMBER               —         */
E$QLEX BY  00081, /* QUEUE LENGTH EXCEEDED           —         */
E$NBUF BY  00082, /* NO BUFFER SPACE                 —         */
E$INWT BY  00083, /* INPUT WAITING                   —         */
E$NINP BY  00084, /* NO INPUT AVAILABLE              —         */
E$DFD  BY  00085, /* DEVICE FORCIBLY DETACHED        —         */
E$DNC  BY  00086, /* DPTX NOT CONFIGURED             —         */
```

```
E$SICM BY  00087, /* ILLEGAL 3270 COMMAND          —         */
E$SBCF BY  00088, /* BAD 'FROM' DEVICE             —         */
E$VKBL BY  00089, /* KBD LOCKED                    —         */
E$VIA  BY  00090, /* INVALID AID BYTE              —         */
E$VICA BY  00091, /* INVALID CURSOR ADDRESS        —         */
E$VIF  BY  00092, /* INVALID FIELD                 —         */
E$VFR  BY  00093, /* FIELD REQUIRED                —         */
E$VFP  BY  00094, /* FIELD PROHIBITED              —         */
E$VPFC BY  00095, /* PROTECTED FIELD CHECK         —         */
E$VNFC BY  00096, /* NUMERIC FIELD CHECK           —         */
E$VPEF BY  00097, /* PAST END OF FIELD             —         */
E$VIRC BY  00098, /* INVALID READ MOD CHAR         —         */
E$IVCM BY  00099, /* INVALID COMMAND               —         */
E$DNCT BY  00100, /* DEVICE NOT CONNECTED          —         */
E$BNWD BY  00101, /* BAD NO. OF WORDS              —         */
E$SGIU BY  00102, /* SEGMENT IN USE                —         */
E$NESG BY  00103, /* NOT ENOUGH SEGMENTS (VINIT$)  —         */
E$SDUP BY  00104, /* DUPLICATE SEGMENTS (VINIT$)   —         */
E$IVWN BY  00105, /* INVALID WINDOW NUMBER         —         */
E$WAIN BY  00106, /* WINDOW ALREADY INITIATED      —         */
E$NMVS BY  00107, /* NO MORE VMFA SEGMENTS         —         */
E$NMTS BY  00108, /* NO MORE TEMP SEGMENTS         —         */
E$NDAM BY  00109, /* NOT A DAM FILE                —         */
E$NOVA BY  00110, /* NOT OPEN FOR VMFA             —         */
E$NECS BY  00111, /* NOT ENOUGH CONTIGUOUS SEGMENTS          */
E$NRCV BY  00112, /* REQUIRES RECEIVE ENABLED      —         */
E$UNRV BY  00113, /* USER NOT RECEIVING NOW        —         */
E$UBSY BY  00114, /* USER BUSY, PLEASE WAIT        —         */
E$UDEF BY  00115, /* USER UNABLE TO RECEIVE MESSAGES         */
E$UADR BY  00116, /* UNKNOWN ADDRESSEE             —         */
E$PRTL BY  00117, /* OPERATION PARTIALLY BLOCKED   —         */
E$NSUC BY  00118, /* OPERATION UNSUCCESSFUL        —         */
E$NROB BY  00119, /* NO ROOM IN OUTPUT BUFFER      —         */
E$NETE BY  00120, /* NETWORK ERROR ENCOUNTERED     —         */
E$SHDN BY  00121, /* DISK HAS BEEN SHUT DOWN       FS        */
E$UNOD BY  00122, /* UNKNOWN NODE NAME (PRIMENET)            */
E$NDAT BY  00123, /* NO DATA FOUND                 —         */
E$ENQD BY  00124, /* ENQUED ONLY                   —         */
E$PHNA BY  00125, /* PROTOCOL HANDLER NOT AVAIL   DPTX       */
E$IWST BY  00126, /* E$INWT ENABLED BY CONFIG     DPTX       */
E$BKFP BY  00127, /* BAD KEY FOR THIS PROTOCOL    DPTX       */
E$BPRH BY  00128, /* BAD PROTOCOL HANDLER (TAT)   DPTX       */
E$ABTI BY  00129, /* I/O ABORT IN PROGRESS        DPTX       */
E$ILFF BY  00130, /* ILLEGAL DPTX FILE FORMAT     DPTX       */
E$TMED BY  00131, /* TOO MANY EMULATE DEVICES     DPTX       */
E$DANC BY  00132, /* DPTX ALREADY CONFIGURED      DPTX       */
E$NENB BY  00133, /* REMOTE MODE NOT ENABLED       NPX       */
E$NSLA BY  00134, /* NO NPX SLAVE AVAILABLE        ——        */
E$PNTF BY  00135, /* PROCEDURE NOT FOUND          R$CALL     */
E$SVAL BY  00136, /* SLAVE VALIDATION ERROR       R$CALL     */
E$IEDI BY  00137, /* I/O error or device interrupt (GPPI)    */
E$WMST BY  00138, /* Warm start happened (GPPI)              */
```

```
       E$DNSK BY  00139, /* A pio instruction did not skip (GPPI)  */
       E$RSNU BY  00140, /* REMOTE SYSTEM NOT UP         R$CALL    */
       E$S18E BY  00141,
/*                                                                 */
/*     New error codes for REV 19 begin here:                      */
/*                                                                 */
       E$NFQB BY  00142, /* NO FREE QUOTA BLOCKS          —        */
       E$MXQB BY  00143, /* MAXIMUM QUOTA EXCEEDED        —        */
       e$max_quota_exceeded
              by    143, /* alias to E$MXQB                        */
       E$NOQD BY  00144, /* NOT A QUOTA DISK (RUN VFIXRAT)         */
       E$QEXC BY  00145, /* SETTING QUOTA BELOW EXISTING USAGE     */
       E$IMFD BY  00146, /* Operation illegal on MFD               */
       E$NACL BY  00147, /* Not an ACL directory                   */
       E$PNAC BY  00148, /* Parent not an ACL directory            */
       E$NTFD BY  00149, /* Not a file or directory                */
       E$IACL BY  00150, /* Entry is an ACL                        */
       E$NCAT BY  00151, /* Not an access category                 */
       E$LRNA BY  00152, /* Like reference not available           */
       E$CPMF BY  00153, /* Category protects MFD                  */
       E$ACBG BY  00154, /* ACL too big                            */
       E$ACNF BY  00155, /* Access category not found              */
       E$LRNF BY  00156, /* Like reference not found               */
       E$BACL BY  00157, /* BAD ACL                                */
       E$BVER BY  00158, /* BAD VERSION                            */
       E$NINF BY  00159, /* NO INFORMATION                         */
       E$CATF BY  00160, /* Access category found (Ac$rvt)         */
       E$ADRF BY  00161, /* ACL directory found (Ac$rvt)           */
       E$NVAL BY  00162, /* Validation error (nlogin)              */
       E$LOGO BY  00163, /* Logout (code for fatal$)               */
       E$NUTP BY  00164, /* No unit table available. (PHANT$)      */
       E$UTAR BY  00165, /* Unit table already returned. (UTDALC)  */
       E$UNIU BY  00166, /* Unit table not in use. (RTUTBL)        */
       E$NFUT BY  00167, /* No free unit table. (GTUTBL)           */
       E$UAHU BY  00168, /* User already has unit table. (UTALOC)  */
       E$PANF BY  00169, /* Priority ACL not found.                */
       E$MISA BY  00170, /* Missing argument to command.           */
       E$SCCM BY  00171, /* System console command only.           */
       E$BRPA BY  00172, /* Bad remote password          R$CALL    */
       E$DTNS BY  00173, /* Date and time not set yet.             */
       E$SPND BY  00174, /* REMOTE PROCEDURE CALL STILL PENDING    */
       E$BCFG BY  00175, /* NETWORK CONFIGURATION MISMATCH         */
       E$BMOD BY  00176, /* Illegal access mode   (AC$SET)         */
       E$BID  BY  00177, /* Illegal identifier    (AC$SET)         */
       E$ST19 BY  00178, /* Operation illegal on pre-19 disk       */
       E$CTPR BY  00179, /* Object is category-protected (Ac$chg)  */
       E$DFPR BY  00180, /* Object is default-protected (Ac$chg)   */
       E$DLPR BY  00181, /* File is delete-protected (Fil$dl)      */
       E$BLUE BY  00182, /* Bad LUBTL entry              (F$IO)    */
       E$NDFD BY  00183, /* No driver for device         (F$IO)    */
       E$WFT  BY  00184, /* Wrong file type              (F$IO)    */
```

```
E$FDMM BY  00185, /* Format/data mismatch         (F$IO)     */
E$FER  BY  00186, /* Bad format                   (F$IO)     */
E$BDV  BY  00187, /* Bad dope vector              (F$IO)     */
E$BFOV BY  00188, /* F$IOBF overflow              (F$IO)     */
E$NFAS BY  00189, /* Top-level dir not found or inaccessible*/
E$APND BY  00190, /* Asynchronous procedure still pending    */
E$BVCC BY  00191, /* Bad virtual circuit clearing            */
E$RESF BY  00192, /* Improper access to a restricted file    */
E$MNPX BY  00193, /* Illegal multiple hops in NPX.           */
E$SYNT BY  00194, /* SYNTanx error                           */
E$USTR BY  00195, /* Unterminated STRing                     */
E$WNS  BY  00196, /* Wrong Number of Subscripts              */
E$IREQ BY  00197, /* Integer REQuired                        */
E$VNG  BY  00198, /* Variable Not in namelist Group          */
E$SOR  BY  00199, /* Subscript Out of Range                  */
E$TMVV BY  00200, /* Too Many Values for Variable            */
E$ESV  BY  00201, /* Expected String Value                   */
E$VABS BY  00202, /* Variable Array Bounds or Size           */
E$BCLC BY  00203, /* Bad Compiler Library Call               */
E$NSB  BY  00204, /* NSB tape was detected                   */
E$WSLV BY  00205, /* Slave's ID mismatch                     */
E$VCGC BY  00206, /* The virtual circuit got cleared.        */
E$MSLV BY  00207, /* Exceeds max number of slaves per user   */
E$IDNF BY  00208, /*  Slave's ID not found                   */
E$NACC BY  00209, /* Not accessible                          */
E$UDMA BY  00210, /* Not Enough DMA channels                 */
E$UDMC BY  00211, /* Not Enough DMC channels                 */
E$BLEF BY  00212, /* Bad tape record length and EOF          */
E$BLET BY  00213, /* Bad tape record length and EOT          */
E$ALSZ BY  00214, /* Allocate request too small              */
E$FRER BY  00215, /* Free request with invalid pointer       */
E$HPER BY  00216, /* User storage heap is corrupted          */
E$EPFT BY  00217, /* Invalid EPF type                        */
E$EPFS BY  00218, /* Invalid EPF search type                 */
E$ILTD BY  00219, /* Invalid EPF LTD linkage descriptor      */
E$ILTE BY  00220, /* Invlaid EPF LTE linkage discriptor      */
E$ECEB BY  00221, /* Exceeding command environment breadth   */
E$EPFL BY  00222, /* EPF file exceeds file size limit        */
E$NTA  BY  00223, /* EPF file not active for this user       */
E$SWPS BY  00224, /* EPF file suspended within program session *

E$SWPR BY  00225, /* EPF file suspended within this process */
E$ADCM BY  00226, /* System administrator command ONLY       */
```

```
E$UAFU BY  00227, /* Unable to allocate file unit              */
e$unable_to_allocate_file_unit
       by  00227, /* alias to E$UAFU                           */
E$FIDC BY  00228, /* File inconsistent data count              */
e$file_inconsistent_data_count
       by  00228,  /* alias to e$fidc                          */
e$indl by  00229,  /* alias to e$insufficient_dam_level        */
e$insufficient_dam_levels
       by  00229, /* Not enough dam index levels as needed     */
e$peof by  00230, /* alias to e$past_EOF                       */
e$past_EOF
       by  00230, /* Past End Of File                          */
E$N231 by  00231, /* Error code 231.                           */
E$N232 by  00232, /* Error code 232.                           */
E$N233 by  00233, /* Error code 233.                           */
E$N234 by  00234, /* Error code 234.                           */
E$N235 by  00235, /* Error code 235.                           */
E$N236 by  00236, /* Error code 236.                           */
E$N237 by  00237, /* Error code 237.                           */
E$N238 by  00238, /* Error code 238.                           */
E$RSHD by  00239, /* Remote disk has been shut down.           */
E$LAST BY  00239; /* THIS ***MUST*** BE. LAST          —       */
/*                                                             */
/*  The value of E$LAST must equal the last error code.        */
/*                                                             */
/**************************************************************/
```

P
Subroutines from MRUs

INTRODUCTION

The following subroutines have already been documented in three earlier Minor Release Updates (MRUs) — for PRIMOS Revisions 19.1, 19.2, 19.3. The MRUs also supply several important corrections to be made to certain subroutines already documented in the Subroutines Reference Guide.

This appendix does not repeat those corrigenda found in the MRUs; instead of being repeated here, they will be inserted within the subroutine documentation undergoing revision for a new edition of the Subroutines Reference Guide as part of PRIMOS Rev. 20 documentation. However, these subroutines are repeated here to enable the user to find all the released subroutines within the present Subroutines Reference Guide or its update.

If the user cannot wait for all addenda and corrigenda to be collated within the Rev. 20 edition of the Subroutines Reference Guide, PRIME urges the user to refer to the corrections in the following MRUs:

MRU4304-009: Revision 19.1 Software Release Document

MRU4304-010: Revision 19.2 Software Release Document

MRU4304-011: Revision 19.3 Software Release Document

The following subroutines were released for Rev. 19.1, announced in the order given:

Subroutine	Function
PAR$RV	Return the revision number of a disk partition.
PRI$RV	Return the revision number of the currently running PRIMOS operating system.
SETRC$	Return the error code to the invoking command processor.
MGSET$	Set the message receive state of the calling process.
MSG$ST	Determine the receive state of the processes for a user.
RMSGD$	Return waiting deferred messages to the caller.
SMSG$	Send a message.

The following subroutines were released for Rev. 19.2, announced in the order given:

Subroutine	Function
SS$ERR	Perform subsystem error handling.
ERTXT$	Accept an error code and return its corresponding error message.
DIR$SE	Perform a directory search, responding to caller-specified criteria.
TTY$IN	Check for characters in a user's TTY buffer.

The following subroutines were released for Rev. 19.3, announced in the order given:

Subroutine	Function
LIMIT$	Set timer(s) within PRIMOS.
PRJID$	Return a user's login project name.

SUBROUTINE DESCRIPTIONS FOR REV. 19.1

The following subroutines were released for Rev. 19.1.

Please insert the following before the entry for PHANT$ on page 10-34:

▶ PAR$RV

Purpose

This function returns the revision number of a disk partition, given the name of the partition.

Usage

```
dcl par$rv entry(char(32) var, fixed bin(15))
        returns (fixed bin(15));

par_rev = $par$rv(part_name, code);
```

part_name 32-character varying string containing the partition name.

code error return code.

E$FNTF	Partition name not found in disk tables.
E$BNAM	Illegal disk partition name.

par_rev partition revision number.

0	Pre-ACLs and quotas.
1	Converted to allow ACLs and quotas.
-1	Error — see error return code (above).

Please insert the following entry before the entry for PWCHK$ on page 10-36:

▶ PRI$RV

Purpose

This subroutine returns the revision number of the currently running PRIMOS operating system.

Usage

```
dcl pri$rv entry(char(32) var);

call pri$rv(primos_rev);
```

primos_rev 32-character varying string containing the PRIMOS revision number.

The following text should be inserted before the description of TEXTO$ on page 10-44:

▶ SETRC$

Purpose

This subroutine permits static mode programs to return an error code value to the command processor that invoked them.

Usage

```
dcl setrc$ entry(fixed bin(15));

call setrc$(errcode);
```

errcode Error code value to be returned to the command processor (input). Zero indicates no error.

Message Subroutines

Please replace the descriptions of the message subroutines in Appendix B with the following information.

▶ MGSET$

Purpose

MGSET$ is used to set the message receive state of the calling process. The receive state determines the willingness of the process to accept messages sent to it. There are three possible states that a process may have: accept all messages, accept only deferred messages, and reject all messages. Messages that are deferred are not necessarily delivered immediately when sent, but rather are buffered by the system and delivered later. Deferring messages allows the receiver to accept messages at times that are convenient for him or her, rather than at times convenient to the sender. Users may explicitly request waiting deferred messages via the RMSGD$ call, or they may allow the system to deliver deferred messages automatically after PRIMOS commands complete their execution.

Usage

```
dcl mgset$ entry(fixed bin(15), fixed bin(15));

call mgset$(key, code);
```

key — Provided by the user. A standard system key that specifies the receive state to be set.

K$ACPT	Accept all messages.
K$DEFR	Accept only deferred messages.
K$RJCT	Reject all messages.

code — A standard system error code returned by the subroutine.

E$BKEY	Bad key.
0	No error.

▶ MSG$ST

Purpose

MSG$ST allows the caller to determine the receive state of processes. If the caller supplies a specific user number, the receive state and user name of that process are returned. If the caller supplies a user name, the user number and receive state of the most permissive user with the specified name are returned.

Usage

```
dcl msg$st entry(fixed bin(15), fixed bin(15), char(*),
           fixed bin(15), char(*), fixed bin(15), fixed bin(15));

call msg$st(key, user_num, system_name, system_name_len,
          user_name, user_name_len, receive_state);
```

key — Provided by the user. Can be either of the following:

K$READ	Return the user's name and state for user user_num on system system_name.
2	Return the user's number and state for user user_name on system system_name.

user_num — The user number of the process. If key = K$READ, user_num is provided by the user. If key = 2, user_num is returned by the subroutine.

system_name — The name of the system on which the desired process is found. Provided by the user.

system_name_len — The length of system_name in characters. If system_name_len = 0, the local system is assumed. Provided by the user.

user_name — The user name of the process. If key = K$READ, this parameter is returned by the subroutine. If key = 2, this parameter is provided by the user.

user_name_len — The length of user_name in characters. Provided by the user.

receive_state — The receive state of the process. This parameter can be any of the following:

K$ACPT	Accepting all messages.
K$DEFR	Accepting deferred messages only.
K$RJCT	Rejecting all messages.
K$NONE	User does not exist.
K$BKEY	Invalid state, bad key in call.
K$BREM	Invalid state, bad system_name.

▶ RMSGD$

Purpose

RMSGD$ returns waiting deferred messages to the caller. This routine does not return immediate messages. Users wishing to obtain all messages via this routine must inhibit immediate messages by setting their receive state to receive only deferred messages (see MGSET$ with a key of K$DEFR).

Usage

```
dcl rmsgd$ entry(char(*), fixed bin(15), fixed bin(15), char(*),
             fixed bin(15), fixed bin(15), char(*), fixed bin(15));

call rmsgd$(from_name, from_name_len, from_num, system_name,
            system_name_len, time_sent, text, text_len);
```

from_name — The user name of the sender. Returned by the subroutine.

from_name_len — The length of from_name in characters. Provided by the user.

from_num — The sender's user number. Returned by the subroutine.

system_name — The name of the system from which the message was sent. Returned by the subroutine.

system_name_len — The length of system_name in characters. Provided by the user.

time_sent — The time, in minutes past midnight, at which the message was sent. If no message is returned, time_sent is set to -1. Returned by the subroutine.

text — The text of the message. Returned by the subroutine.

text_len — The length of text. Provided by the user.

▶ SMSG$

Purpose

SMSG$ sends a message. Messages may either be sent immediately or deferred. Immediate messages are delivered to the recipient at the time the message is sent. Deferred messages are held in a system buffer until the receiver requests them. (Deferred messages are also delivered to a user automatically after each PRIMOS command completes execution.) Messages may be sent to other processes by addressing them to either their user numbers or their user names. If user name is used, all interactive users with that name will receive the message.

Usage

```
dcl smsg$ entry(fixed bin(15), char(*), fixed bin(15),
                fixed bin(15), char(*), fixed bin(15), char(*),
                fixed bin(15), (131) fixed bin(15));
```

```
call smsg$(key, to_name, to_name_len,
           to_user_num, to_system_name, to_system_len, text,
           text_len, error_vector);
```

All parameters except error_vector are provided by the user.

key — Specifies the type of message, immediate or deferred.

- 0 — Deferred message. Messages are buffered and delivered at the receiver's convenience.
- 1 — Immediate message. Messages are delivered immediately when sent.

to_name — The user name of the user to whom the message is to be sent. If to_name is nonblank, the message is sent to all interactive users logged in under that name. If to_name is blank, the message is sent by to_num, and to_name is ignored.

to_name_len — The length of to_name in characters.

to_user_num — The user number of the user to whom the message is sent. If to_num is positive, to_name is ignored. If to_num is zero and to_name is blank, the message is sent to the operator.

to_system_name
The name of the node to which the message is to be sent.

to_system_len
The length of to_system_name in characters. If to_system_len is zero, the local system is assumed.

text
The text of the message. Messages may be up to 80 characters in length, and either blank-padded or terminated with a linefeed. Only printable characters and the bell character are printed by the operating system.

text_len
The length of text in characters.

error_vector
An array that reports the success or failure of the call. Its size can range from 4 through 131. Its elements have the following meanings:

error_vector (1)
An overall status code returned by the subroutine.

E$NRCV	Operation aborted because sender does not have receive enabled.
E$UADR	Unknown addressee.
E$UDEF	Receiver not receiving.
E$PRTL	Operation partially blocked.
E$NSUC	Operation failed.
0	Operation succeeded.

error_vector (2)
Three less than the total number of elements in error_vector. Normally set to the number of configured users (128). Provided by the user.

error_vector (3)
An overall network error code returned by the subroutine.

XS$CLR	Connect cleared.
XS$BPM	Unknown node address.
XS$DWN	Node not responding.

error_vector (4-131)
An optional status vector whose length is the value of error_vector (2). If supplied, each element is a status code returned by the subroutine, indicating success or failure to send a message to user number n-3, where n is the index

into error_vector. For example, error_vector (10) is the status for user number 7.

E$UBSY	User busy, please wait.
E$UNRV	User not receiving now.

SUBROUTINE DESCRIPTIONS FOR REV. 19.2

Add the following new subroutines to Chapter 10:

▶ TTY$IN

Purpose

This function checks whether there are any characters in the user's TTY input buffer. The state of the buffer is undisturbed by the call; no character is actually read or removed from the buffer.

Usage

```
dcl tty$in entry() returns (bit(1)aligned);

more-to-read = tty$in;
```

more-to-read Will be true ('1'b) if there is at least one character of input available at the terminal of the calling process, and '0'b otherwise.

Discussion

TTY$IN is used to check whether there is at least one character of input currently available on the calling process' terminal. Use TTY$IN when you do not want to wait for input via a call to CL$GET, C1IN$, or T1IN. TTY$IN allows the program to poll for input and perform other processing while waiting for input to arrive.

If TTY$IN is called in a noninteractive process, '0'b is always returned, whether or not a command input file is active.

It is possible for TTY$IN to return '1'b, and for a subsequent call to C1IN$ to wait for input. This can happen if the user types Control-P after TTY$IN is called, which causes a quit to PRIMOS and the flushing of the input buffer. When the user types START, the next call to C1IN$ will then wait for a character.

TTY$IN is necessary at Rev. 19 to cut down on CPU usage. Before Rev. 19, checks of the input buffer could be done only with an R-mode routine that, at Rev. 19, has a high overhead of CPU usage. Use of TTY$IN can cut CPU usage by half.

Because FTN cannot call subroutines with no argument, this routine may not be called directly from FTN. To get the benefits of the routine, use an F77 or PMA interlude.

Command Error Reporting

This discussion applies to the two subroutines that follow, SETRC$ and SS$ERR.

When a command or subsystem detects an error situation, two parties must in general be notified: the user, who is usually interactive, and the invoker, which is simply the procedure that invoked the command or subsystem. Typically, the user is notified by means of a diagnostic message, whereas the invoker must be notified by a method more suitable for programmed decisions--a status code.

The requirement that subsystems be able to keep control on errors if interactive but give up control if noninteractive is met by requiring subsystems to call the routine SS$ERR. Use of SS$ERR is necessary to support the Command Procedure Language product. Without it, CPL is not able to support its documented error handling features fully because it does not receive proper indication of compilation, loading, and file handling failures.

Severity Codes

A severity code is a single FIXED BINARY(15,0) value in which two distinct pieces of information may be encoded. First, the severity level has the value 0, -1, or +1; this is the arithmetic sign of the severity code. Second, the absolute value of the severity code may (or may not) be a standard error code, as defined below. The meaning, if any, of the absolute value of a severity code must be interpreted relative to the specific command that returned it: the same absolute value returned by two different commands may not mean the same thing.

The meanings of the severity level of the severity code, however, are the same no matter which command returned the code. They are as follows:

0 No errors--execution successful.

-1 Warning(s)--minor exceptions encountered, but the results of the command's execution are usable to the best of the command's ability to determine.

1 Error(s)—serious errors encountered. Some of the results of the command are not usable, or some of the actions requested could not be performed.

When a command or command function has decided to return control to its caller, it must also return a severity code value if it encountered an error. Command callers initialize the severity code to 0 before calling a command so that the command need take no action if no errors are encountered.

If the procedure is part of a user-created program, it should use the primitive SETRC$ to return the severity code.

Standard Error Codes

A standard error code is always to be interpreted according to some error table. Error tables are identified by 32-character names. At present, only the PRIMOS error table exists, accessible via ERRPR$. It is assigned the null name. See Appendix E, Error Handling for I/O Subroutines.

A standard error code is a compact representation of a diagnostic message and is usually returned by a command or subroutine to its caller. This code identifies the precise cause of an error encountered by the callee. A standard error code is converted to a severity code by changing its arithmetic sign to the proper severity level value.

Subsystem Error Handling

Whenever a conversational subsystem encounters an error in the syntax of a subcommand or during its execution and that subsystem wishes to returns to its own command level, it must:

1. Print any applicable diagnostics;

2. Call the PRIMOS subroutine SS$ERR (subsystem error);

3. Return to its command level;

4. Not return a positive severity code when it finally returns control to PRIMOS, since then the user would see an ER! prompt when he is not expecting one.

When a subsystem encounters an error and immediately returns to PRIMOS without going back to its own command level, it does not make any difference whether the subsystem is being used interactively or not. Hence the subsystem should:

1. Print any applicable diagnostics;
2. Call SETRC$ to set a positive (or negative) severity code as appropriate;
3. Return to PRIMOS. The user will see an ER! prompt, if interactive, or a CPL procedure will receive the proper error code, if not.
4. SS$ERR should not be called in this case.

SS$ERR works approximately as follows. When called, SS$ERR checks whether the user is interactive, that is, whether the process is a non-phantom whose command input stream is connected to the terminal. If so, SS$ERR simply returns. Otherwise, SS$ERR raises the condition SUBSYS_ERR$. The default handling of this condition is for the command processor to abort the subsystem via a nonlocal goto back into the command processor, where a positive severity code is forced.

Users and subsystem implementors should keep the following points in mind:

- The user's program may make an on-unit for SUBSYS_ERR$, which simply returns. This causes SS$ERR to return to the subsystem as if the user were interactive, thus defeating the noninteractive abort mechanism. (This option would rarely be useful.)
- The subsystem may use the condition mechanism's CLEANUP$ condition to regain control in one last gasp before the nonlocal goto is completed. (For details on the condition mechanism, see Chapter 22.) This will allow the subsystem to perform any required cleanup activities before it actually loses control.

Subsystems should call SS$ERR after printing diagnostics and before returning to their command level if they intend to retain control.

Subsystems should not call SS$ERR if they will return to PRIMOS immediately on the error.

Calling Sequences for Subroutines Affected

▶ SETRC$

(SETRC$ was released for Revision 19.1. Its calling sequence is described earlier in this appendix.)

▶ SS$ERR

Purpose

This subroutine is used for subsystem error handling as discussed above. If the caller is being used interactively, SS$ERR simply returns. Otherwise, the condition SUBSYS_ERR$ is raised, which usually results in the termination of the caller by means of a nonlocal goto back to the command processor.

Usage

```
dcl ss$err entry();

call ss$err;
```

▶ ERTXT$

Purpose

This routine accepts a standard PRIMOS error code and returns the character string representation of its error message as it would be printed by the routine ERRPR$.

Usage

```
dcl ertxt$ entry(fixed bin, char(1024)var);

call ertxt$(code, errmsg);
```

code	Standard error code. (Input)
errmsg	Text of error message. (Output)

Add the following to page A-28.

▶ DIR$SE

Purpose

This new routine replaces and extends the functionality of DIR$LS. DIR$SE is a general purpose directory searcher that returns entries meeting caller-specified selection criteria.

Usage

```
dcl dir$se entry  (fixed bin(15), fixed bin(15), bit(1), ptr,
                  ptr, fixed bin(15), fixed bin(15),
                  fixed bin(15), (4) fixed bin(15), fixed bin(15),
                  fixed bin(15));

call dir$se  (dir_unit, dir_type, initialize, sel_ptr,
             return_ptr, max_entries, entry_size,
             ent_returned, type_counts, max_type,
             code);
```

dir_unit — Unit on which directory to be searched is open. (Input)

dir_type — Type of object open on dir_unit. (Input)

initialize — If set, directory is to be reset to the beginning. If unset, it is to be searched from the current position. (Input)

sel_ptr — Pointer to structure containing selection criteria (see below). (Input)

return_ptr — Pointer to caller's return structure for selected entry data (see below). (Input)

max_entries — Maximum number of entries to be returned (should be greater than zero unless this routine is being used only to initialize the directory). (Input)

entry_size — Number of words to be returned per entry. (Input)

ent_returned — Number of entries returned. (Output)

type_counts — Number of entries of each type returned in the order: dirs, seg dirs, files, access categories. (This argument should be an array of 4 halfwords.) The type_counts are incremented each time DIR$SE is called, that is, the number of types returned in this call of DIR$SE is added to the current type-counts totals. When the "initialize" bit is set, these counts are reset to the total number of types returned in this call. (Input/output)

max_type — Number of types in type_counts (currently must be 4). (Input)

code — Standard error code. (Output)
Possible values are:

e$bver — Bad version number for selection criteria structure (currently can only be zero (0)).

e$bpar — Bad max_type (currently must be 4).

e$eof — There are no more entries in the directory to be selected.

e$st19 — Selection criteria involving date/time last saved or RBF file type have been specified, and the PRIMOS rev that accesses the directory does not support these features.

The selection criteria should be supplied in the following structure. The "sel_ptr" parameter should point to this structure.

```
dcl 1 selection_criteria based,
      2 version_no fixed bin,
      2 wild_ptr ptr,
      2 wild_count fixed bin (15),
      2 desired_types,
        3 dirs bit(1),
        3 seg_dirs bit(1),
        3 files bit(1),
        3 access_cats bit(1),
        3 RBF bit(1),
        3 spare bit(11),
      2 modified_before_date_time fixed bin (31),
      2 modified_after_date_time fixed bin (31),
      2 saved_before_date_time fixed bin (31),
      2 saved_after_date_time fixed bin (31),
```

where:

version_no — Must be zero for this version of the selection criteria structure.

wild_ptr — If wildcard entryname selection is to be applied to the directory entries, this field should point to a list of wildcard names for which to search. The list should be an array of char(32) varying strings, and the names must be in uppercase. Wildcards are explained in the Prime User's Guide.

wild_count — Is the number of names in the list pointed to by wild_ptr. If wild_count is zero, entryname is not used as a selection criterion.

desired_types — A bit-encoded field defining which types of directory entries the caller wishes to have returned. The first four bits of this field specify the physical types of the entries that are to be returned. The fifth bit can be used in combination with the other four bits to select entries that are also RBF entries, and thus have a logical type of '1'.

To select only RBF segment directories, the seg_dirs and RBF bits should both be set, and the other bits not set. If the first four bits are set, all entries will be returned. If all five bits are set, all entries that are also RBF entries will be returned.

modified_before_date — Selects entries with date/time modified earlier than this date. The date should be in standard FS format (described with routine CV$DQS). Should be zero if this field is not to be used as a selection criterion.

modified_after_date — Selects entries with date/time modified later than this date. The date should be in standard FS format (described with routine CV$DQS). Should be zero if this field is not to be used as a selection criterion.

saved_before_date — Reserved for future use. Must be zero currently.

saved_after_date — Reserved for future use. Must be zero currently.

DIR$SE will return the information for all the entries selected by this call in the following structure:

```
dcl 1 dir_entries (*) based,
      2 ecw,
        3 type bit (8),
        3 length bit (8),
      2 entryname char(32) var,
      2 protection,
        3 owner rights,
          4 spare bit (5),
          4 delete bit (1),
          4 write bit (1)
          4 read bit (1),
        3 delete_protect bit (1),
        3 non_owner_rights,
          4 spare bit (4),
          4 delete bit (1),
          4 write bit (1),
          4 read bit (1),
      2 file_info,
        3 long_rat_hdr bit(1),
        3 dumped bit(1),
        3 dos_mod bit (1),
        3 special bit (1),
        3 rwlock bit (1),
        3 spare bit (2),
        3 type bit (8),
      2 date_time_mod fixed bin (31),
      2 non_default_acl bit (1) aligned,
      2 logical_type fixed binary,
      2 trunc bit (1) aligned,
      2 date_time_last_saved fixed bin (31);
```

where:

ecw — Entry control word for the entry:

type:	2	normal directory entry (file, directory, or segment directory).
	3	access category.
length:	24	words for PRIMOS revs up to and including 19.2; 27 words for PRIMOS revs from 19.2 onwards.

entryname — Name of the entry.

protection — owner_rights — These are the rights granted to a user when attached to the containing directory with owner rights.

delete_protect — If this bit is set, the file may not be deleted. The bit may be reset by a call to the SATR$$ routine.

non_owner_rights These are the rights granted to a user when attached to the containing directory with non-owner rights.

file_info

long_rat_hdr If set, indicates that the file is a Disk Record Availability (DSKRAT) file spanning more than one disk record.

dumped If set, this file has been saved by MAGSAV and has not been modified since then.

dos_mod If set, this file was modified while PRIMOS II (DOS) was running. It indicates that the date/time last modified field may be incorrect.

special If set, this is a special file (e.g., DSKRAT, BOOT, MFD) and may not be deleted.

rwlock Indicates the setting of the file's read/write concurrency lock.

Possible values are:

0 system default setting
1 unlimited readers or one writer (exclusive)
2 unlimited readers and one writer (update)
3 unlimited readers and writers (none)

type Indicates the type of object described by this entry. Possible values are:

0 SAM file
1 DAM file
2 SAM segment directory
3 DAM segment directory
4 UFD
6 Access category

date_time_mod The date/time the file was last modified, in standard FS format. FS format dates are described with routine CV$DQS.

non_default_acl — This bit is set if the object is not protected by the default ACL — that is, if it is protected by a specific ACL or by an access category.

logical_type — This is an additional file type to the physical file type described in file_info.type. Possible values are:

0 for normal files
1 for RBF files

trunc — This bit is set if the file has been truncated by the FIX_DISK utility; otherwise, reset to zero.

date_time_last_saved — Reserved for future use. This field will currently be returned as zero (unset).

SUBROUTINE DESCRIPTIONS FOR REV. 19.3

Add the following new subroutines to Chapter 10.

▶ LIMIT$

Purpose

LIMIT$ allows the setting of various timers within PRIMOS, each generating a signal if expired. The timer values may also be read.

Usage

```
dcl limit$ entry(fixed bin(15), fixed bin(31), fixed bin(15),

fixed bin(15));

call limit$(key,limit,res,code);
```

key — This key is split into two 8-bit functions. The right half is as follows:

1 = read limit
2 = set limit

The left half is as follows:

1 = cpu limit in seconds
2 = login limit in minutes
5 = CPU watchdog in seconds
6 = real time watchdog in minutes

limit — This is the time to be set in minutes or seconds.

res — Reserved — must be zero.

code — This is a returned standard error code. Refer to Appendix D for a complete listing.

▶ PRJID$

Purpose

This subroutine supports the User Registration and Profiles system. It is intended for use by external login programs that wish to obtain the user's project name. It returns the user's login project name in project_id_name. If the user is logged into the default project, the returned name is DEFAULT.

Usage

```
dcl prjid$ entry(char(32) var);
```

```
call prjid$(project_id_name);
```

Trailing blanks on the project name are not returned. This subroutine is not available in R mode.

Example

```
OK, SLIST PROJECTCALL.F77
      INTEGER*2 PROJECT(17)
      CALL PRJID$(PROJECT)
      CALL TNOU(PROJECT(2),PROJECT(1))
      CALL EXIT
      END
```

```
OK, F77 PROJECTCALL
[F77 Rev. 19.2]
0000 ERRORS [<.MAIN.> F77-REV 19.2}
OK, SEG -LOAD
[SEG rev 19.3.3]
$ LO PROJECTCALL
$ LI
LOAD COMPLETE
$ EXEC
DEFAULT
```

SX
Index of Subroutines by Name

E$xy series	FORTRAN compiler exponentiation routines.	G-8
EDAT$A	Today's date, European (military) style.	12-23
ENCD$A	Make a number printable if possible.	12-23
ENT$RD	Read directory entry with given name.	A-28
EPF$AL	Synonym for EPF$ALLC.	L-13
EPF$ALLC	Allocate storage for EPF's linkage and static date areas.	L-13
EPF$CP	Synonym for EPF$CPF.	L-14
EPF$CPF	Return state of command processor flags in an EPF.	L-14
EPF$DEL	Deactivate one activation of an EPF for the calling process.	L-15
EPF$DL	Synonym for EPF$DEL.	L-15
EPF$INIT	Perform EPF linkage initialization.	L-17
EPF$INVK	Begin execution of a program EPF.	L-18
EPF$MAP	Map procedure images of EPF file into virtual memory.	L-21
EPF$MP	Synonym for EPF$MAP.	L-21
EPF$NT	Synonym for EPF$INIT.	L-17
EPF$RN	Synonym for EPF$RUN.	L-24
EPF$RUN	Perform all appropriate calls to execute an EPF file.	L-24
EPF$VK	Synonym for EPF$INVK.	L-18
EQUAL$	Generate new name for an object name.	M-3
ERKL$$	Read or set erase and kill characters.	10-23
ERRPR$	Interpret a return code.	10-24,D-7
ERRSET	Obsolete error handling.	E-2
ERTXT$	Accept an error code and return its corresponding error message.	P-14
EX$CLR	Disable signalling of EXIT$ condition.	L-28
EX$RD	Return state of counter controlling EXIT$ condition.	L-28
EX$SET	Enable signalling of EXIT$ condition.	L-29
EXIT	Return to PRIMOS.	10-25
EXST$A	Check for file existence.	12-25

F$xxxxx	FORTRAN internal support subroutines.	APPF
F$xxyy series	FORTRAN compiler floating-point functions.	APPG
FDAT$A	Convert the DATMOD field returned by RDEN$$ to DAY MON DD YYYY.	12-25
FEDT$A	Convert the DATMOD field returned by RDEN$$ to DAY DD MON YYYY.	12-26
FIL$DL	Delete a file.	A-29
FILL$A	Fill a string with a character.	12-27
FNCHK$	Check a filename for valid format.	10-25
FORCEW	Write to disk immediately.	9-15
FRE$RA	Release EPF space, returning information via command functions.	L-29
FSUB$A	Fill a substring with a given character.	12-27
FTIM$A	Convert the TIMMOD field returned by REDN$$.	12-28

Q$READ	Read quota information.	10-25
Q$SET	Set quota max.	10-27
QUICK	Partition exchange sort.	13-34
RADXEX	Radix exchange sort.	13-34
RAND$A	Generate random number and update seed, using 32-bit word size and the linear congruential method.	12-43
RD$CED	Synonyn for RD$CE_DP.	L-30
RD$CE_DP	Return current value of the command environment breadth.	L-30
RDASC	Read ASCII from any device.	16-3
RDBIN	Read binary from any device.	16-4
RDEN$$	Position in or read from a UFD.	9-28
RDLIN$	Read a specified number of ASCII characters.	9-35
RDTK$$	Parse a PRIMOS command line.	10-37
RECYCL	Pass control to next user.	10-43
REST$$	Read an R-mode runfile.	9-36
RESU$$	Restore and execute an R-mode runfile.	9-37
RLSE$S	Get input records after SETU$.	13-17,13-21
RMSGD$	Receive a deferred message.	P-6
RNAM$A	Prompt, read a pathname, and check format.	12-44
RNDI$A	Initialize random number generator seed.	12-45
RNUM$A	Prompt and read a number (in any format).	12-46
RPL$	Replace an EPF file with another.	L-30
RPOS$A	Return position of file.	12-47
RRECL	Read disk record (obsolete).	17-5
RSEGAC$	Identify user's access rights to segment.	M-6
RSTR$A	Rotate string left or right.	12-47
RSUB$A	Rotate substring left or right.	12-50
RTRN$S	Get sorted records.	13-17,13-23
RVON$F	Cancel an on-unit from FTN or F77.	22-21
RVONU$	Cancel an on-unit from PL1G or PMA.	22-22
RWND$A	Reposition file.	12-51
S$xy series	FORTRAN compiler subtraction routines.	APPG
SATR$$	Set or modify a file's attributes.	9-38
SAVE$$	Save an R-mode runfile.	9-42
SCHAR	Store a character in an array.	10-43
SEM$CL	Close named semaphore.	21-23
SEM$DR	Drain semaphore.	21-21
SEM$NF	Notify semaphore.	21-19
SEM$OP	Open semaphore by name.	21-16
SEM$OU	Open semaphore by file unit.	21-16
SEM$TN	Set timer for numbered semaphore.	21-22
SEM$TS	Test counter for semaphore.	21-20
SEM$TW	Timed wait for named semaphore.	21-23
SEM$WT	Wait on semaphore.	21-19
SETRC$	Return error code to invoking command processor.	P-4

Update Package

UPD3621-31A

for

SUBROUTINES REFERENCE GUIDE, DOC3621-190

April 1985

This Update Package, UPD3621-31A, is Update 1 for the Third Edition of the SUBROUTINES REFERENCE GUIDE, DOC3621-190. This document should be inserted at the end of your book as Appendix L through Appendix P. Pages that have been added are listed on the next page.

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

The information contained on these updated pages is subject to change without notice and should not be construed as a commitment by Prime Computer Corporation. Prime Computer Corporation assumes no responsibility for any errors that may appear in this package.

Update Package, UPD3621-31A

Pages to change:

Insert Pages	After Pages
Title through x	Coversheet
L-1 through L-36	K-9
M-1 through M-7	L-36
N-1 through N-14	M-7
O-1 through O-14	N-14
P-1 through P-22	O-14

Replace Pages	With Pages
SX-1 through SX-8	SX-1 through SX-10

INDEX