Prime®

Subroutines Reference II: File System

Revision 23.0

DOC10081-2LA

Subroutines Reference II: File System

Second Edition

Sonya Zegarra

This manual documents the software operation of the PRIMOS operating system on 50 Series computers and their supporting systems and utilities as implemented at Master Disk Revision Level 23.0 (Rev. 23.0).

Prime Computer, Inc., Prime Park, Natick, Massachusetts 01760

Printing History

First Edition (DOC10081–1LA) August 1986 for Revision 20.2
Update 1 (UPD10081–11A) July 1987 for Revision 21.0
Update 2 (UPD10081–12A) August 1988 for Revision 22.0
Update 3 (in RLN10247–1LA) July 1989 for Revision 22.1
Second Edition (DOC10081–2LA) June 1990 for Revision 23.0

Credits

Editorial: Barbara Bailey
Index Development: Mary Skousgaard
Illustration: Elizabeth Wahle
Technical Support: Julie Cyphers
Production: Judy Gordon

How to Order Technical Documents

To order copies of documents, or to obtain a catalog and price list:

United States Customers

Call Prime Telemarketing,
toll free, at 1-800-343-2533,
Monday through Thursday,
8:30 a.m. to 8:00 p.m. and
Friday, 8:30 a.m. to 6:00 p.m. (EST).

International

Contact your local Prime
subsidiary or distributor.

PRIME SERVICE^{SM}

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

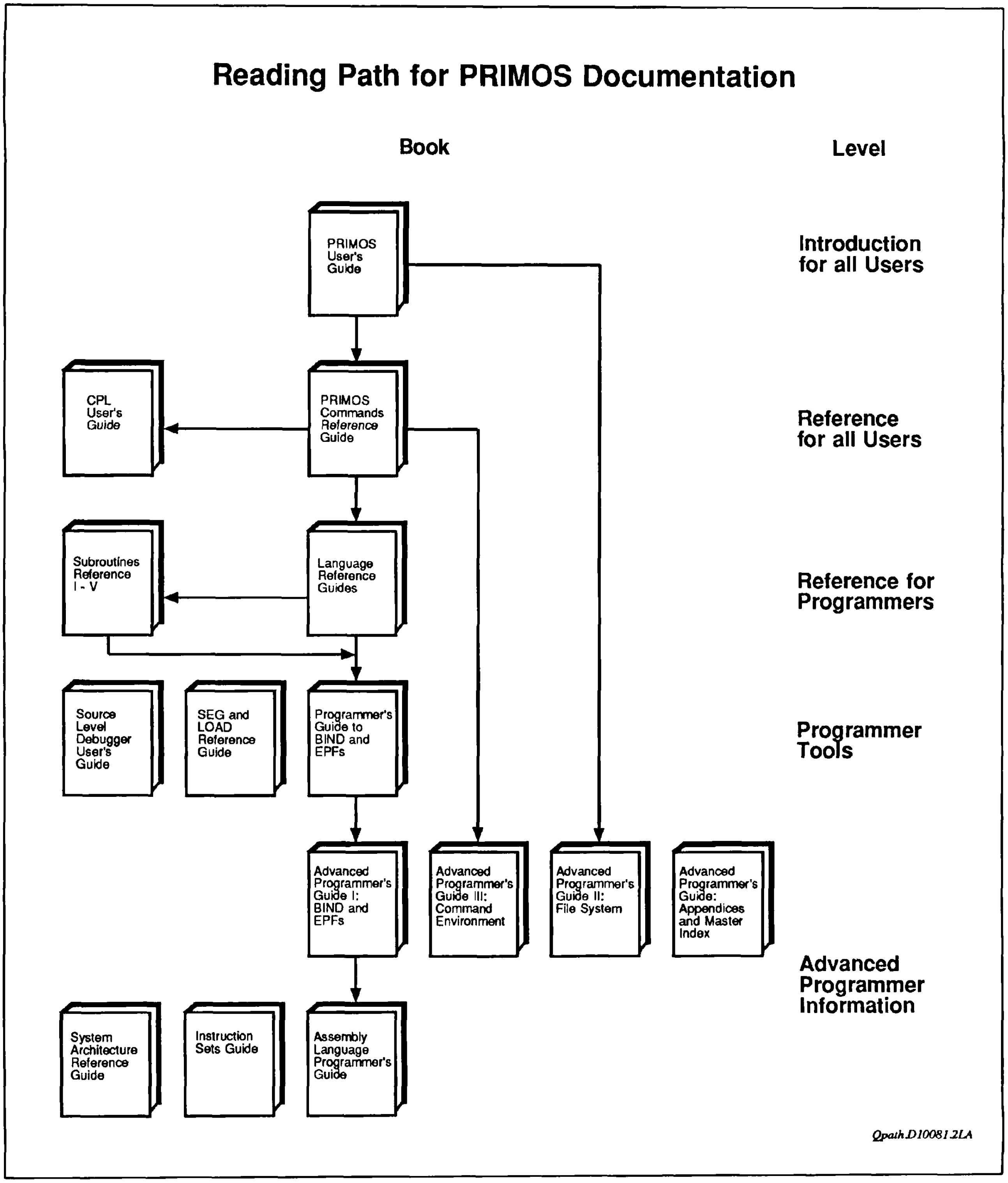
1-800-800-PRIME

For other locations, contact your Prime representative.

Surveys and Correspondence

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701



Reading Path for PRIMOS Documentation

Book
Level
PRIMOS User's Guide
Introduction for all Users
CPL User's Guide
PRIMOS Commands Reference Guide
Reference for all Users
Subroutines Reference I - V
Language Reference Guides
Reference for Programmers
Source Level Debugger User's Guide
SEG and LOAD Reference Guide
Programmer's Guide to BIND and EPFs
Programmer Tools
Advanced Programmer's Guide I: BIND and EPFs
Advanced Programmer's Guide III: Command Environment
Advanced Programmer's Guide II: File System
Advanced Programmer's Guide: Appendices and Master Index
Advanced Programmer Information
System Architecture Reference Guide
Instruction Sets Guide
Assembly Language Programmer's Guide
Qpath.D10081.2LA

Contents

About This Book

Part I: Introduction

1 Overview of Subroutines

Functions and Subroutines . . . 1–1

Subroutine Descriptions . . . 1–2

Subroutine Usage . . . 1–2

Subroutine Parameters . . . 1–5

Part II: File Access and Management Subroutines

2 Access Control

AC$CAT . . . 2–3

AC$CHG . . . 2–5

AC$DFT . . . 2–7

AC$LIK . . . 2–9

AC$LST . . . 2–11

AC$RVT . . . 2–13

AC$SET . . . 2–14

CALAC$ . . . 2–16

CAT$DL . . . 2–18

GETID$ . . . 2–19

GPAS$$ . . . 2–21

ISACL$ . . . 2–23

PA$DEL . . . 2–24

PA$LST . . . 2–25
PA$SET . . . 2–27
SPAS$$ . . . 2–29

3 Attaching

AT$ . . . 3–3
AT$ABS . . . 3–6
AT$ANY . . . 3–9
AT$HOM . . . 3–11
AT$LDEV . . . 3–13
AT$OR . . . 3–15
AT$REL . . . 3–17
AT$ROOT . . . 3–19
ATCH$$ . . . 3–20
GTROB$ . . . 3–21

4 File and Directory Manipulation

APSFX$ . . . 4–4
CH$MOD . . . 4–6
CL$FNR . . . 4–7
CLO$FN . . . 4–9
CLO$FU . . . 4–10
CNAM$$ . . . 4–11
CREA$$ . . . 4–13
CREPW$ . . . 4–14
DIR$CR . . . 4–15
DIR$LS . . . 4–17
DIR$RD . . . 4–24
DIR$SE . . . 4–29
ENT$RD . . . 4–38
EQUAL$ . . . 4–40
EXTR$A . . . 4–41
FIL$DL . . . 4–43
FINFO$ . . . 4–45
FNCHK$ . . . 4–49
FORCEW . . . 4–51
GPATH$ . . . 4–53
ISREM$ . . . 4–56

LDISK$ . . . 4–58
LUDSK$ . . . 4–61
NAM$AD_PORTAL . . . 4–63
NAM$L_GMT . . . 4–65
NAM$RM_PORTAL . . . 4–68
PAR$RV . . . 4–69
PRWF$$ . . . 4–71
Q$READ . . . 4–79
Q$SET . . . 4–82
RDEN$$ . . . 4–84
RDLIN$ . . . 4–85
SATR$$ . . . 4–87
SGD$DL . . . 4–92
SGD$EX . . . 4–93
SGD$OP . . . 4–94
SGDR$$ . . . 4–96
SIZE$ . . . 4–102
SRCH$$ . . . 4–105
SRSFX$ . . . 4–114
TNCHK$ . . . 4–121
TSRC$$ . . . 4–123
UNITS$ . . . 4–124
WILD$ . . . 4–125
WTLIN$ . . . 4–126
CAM File Subroutines . . . 4–128
CF$EXT . . . 4–132
CF$REM . . . 4–134
CF$SME . . . 4–137

Part III: EPF Management Subroutines

5 *EPF Management*

EPF$ALLC . . . 5–3
EPF$CPF . . . 5–5
EPF$DEL . . . 5–7
EPF$INIT . . . 5–9
EPF$INVK . . . 5–11

EPF$ISREADY . . . 5–15

EPF$MAP . . . 5–17

EPF$REG . . . 5–20

EPF$RUN . . . 5–22

EPF$UREG . . . 5–25

LN$SET . . . 5–27

REMEPF$ . . . 5–29

RPL$ . . . 5–31

Part IV: Command Environment Subroutines

6 Command Environment

CE$BRD . . . 6–2

CE$DPT . . . 6–3

CL$PIX . . . 6–4

CP$ . . . 6–8

GV$GET . . . 6–11

GV$SET . . . 6–13

LIST$CMD . . . 6–15

LV$GET . . . 6–17

LV$SET . . . 6–19

RD$CE_DP
RD$CED . . . 6–21

Part V: Search Rules Subroutines

7 Search Rules

OPSR$ . . . 7–3

OPSRS$ . . . 7–9

SR$ABSDS . . . 7–16

SR$ADDB . . . 7–19

SR$ADDE . . . 7–22

SR$CREAT . . . 7–25

SR$DEL . . . 7–27

SR$DSABL . . . 7–29

SR$ENABL . . . 7–32

SR$EXSTR . . . 7–35

SR$FR_LS . . . 7–39

SR$INIT . . . 7–41

SR$LIST . . . 7–43

SR$NEXTR . . . 7–47

SR$READ . . . 7–52

SR$REM . . . 7–56

SR$SETL . . . 7–59

SR$SSR . . . 7–62

Appendices

A Obsolete File System Subroutines

ATCH$$ . . . A–2

CREA$$ . . . A–5

CREPW$ . . . A–7

RDEN$$ . . . A–8

TSRC$$ . . . A–17

B Data Type Equivalents

C Argument Parsing by the CL$PIX Subroutine

Overview . . . C–1

CL$PIX Operating Modes . . . C–1

Indexes

Index of Subroutines by Function

Access Category . . . FX–2

Access Server Names . . . FX–2

Arrays . . . FX–3

Asynchronous Lines . . . FX–3

Attach Points . . . FX–3

Binary Search . . . FX–4

Buffer Output . . . FX–4

Command Environment . . . FX–4

Command Level . . . FX–5

Condition Mechanism . . . FX–5

Controllers, Asynchronous, Multi-line . . . FX–6

Data Conversion . . . FX-6
Date Formats . . . FX-6
Devices, Assigning or Attaching . . . FX–7
Disk I/O . . . FX–7
Drivers, Device-independent . . . FX–7
Encryption, of Login Password . . . FX–7
EPFs . . . FX–8
Error Handling, I/O . . . FX–9
Event Synchronizers and Event Groups . . . FX–9
Executable Images . . . FX–11
EXIT$ Condition . . . FX–11
File System Objects . . . FX–11
ISC . . . FX–14
Keyboard or ASR Reader . . . FX–15
Logging . . . FX–15
Matrix Operations . . . FX–15
Memory . . . FX–16
Message Facility . . . FX–17
Numeric Conversions . . . FX–17
Paper Tape . . . FX–17
Parsing . . . FX–18
Peripheral Devices . . . FX–18
Phantom Processes . . . FX–19
Process Suspension . . . FX–20
Query User . . . FX–20
Randomizing . . . FX–20
Search Rules . . . FX–20
Semaphores . . . FX–21
Sorting . . . FX–22
Strings . . . FX–23
System Administration . . . FX–24
System Information . . . FX–25
Timers . . . FX–26
User Information . . . FX–27
User Terminal . . . FX–28

Index of Subroutines by Name

Index

About This Book

The *Subroutines Reference* series describes the standard Prime® subroutines and subroutine libraries. Each standard subroutine library is a file containing subroutines that perform a variety of related programming tasks. Whenever these tasks are to be performed, programmers can call the appropriate subroutines in the standard libraries instead of writing their own subroutines. Programmers need to write subroutines only to perform specialized tasks for which no standard subroutines exist.

Overview of This Series

The *Subroutines Reference* consists of five volumes. A brief summary of the contents of each volume follows.

Volume I: Using Subroutines

Volume I introduces the *Subroutines Reference* series. It describes the nature and functions of the Prime standard subroutines and subroutine libraries. It explains how subroutines can be called from programs written in the Prime programming languages: C, COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, PL/I, BASIC V/M, and PMA.

Volume II: File System

Volume II describes subroutines that deal with the access to and management of file system entities, the manipulation of EPFs in the execution environment, system search rules, and the use of a number of command environment functions.

Volume III: Operating System

Volume III describes system subroutines. The subroutines covered in this volume are general system calls to the operating system and the standard system library. These include subroutines for system and user IDs and status, along with the System Information and Metering (SIM) routines. This volume also includes calls for terminal I/O, memory allocation, and program control. Data conversion routines, error message and condition handling routines, semaphores, and an interuser message facility are all found in this volume. An appendix to Volume III lists PRIMOS® standard conditions.

Volume IV: Libraries and I/O

Volume IV presents several mature libraries: the Input/Output Control System (IOCS) library and other I/O-related subroutines, the Application library, the Sort libraries, the FORTRAN Matrix library (MATHLIB), and the CONFIG_USERS library.

IOCS provides device-independent I/O. The chapters on IOCS provide descriptions of the device-independent subroutines plus those device-dependent subroutines simplified by IOCS. Another section provides descriptions of the synchronous and asynchronous device-driver subroutines.

Sections on the Application library, the Sort libraries, and the FORTRAN Matrix library provide descriptions of other program development subroutines especially useful for FORTRAN programs.

The section on CONFIG_USERS describes the subroutines available to the System Administrator who wants to create tailor-made administration programs. CONFIG_USERS replaced EDIT_PROFILE at Rev. 22.1.

Volume V: Event Synchronization

Volume V describes event synchronization and two facilities that use event synchronization: the Timers facility and the InterServer Communications (ISC) facility.

- Event synchronization is made possible by event synchronizers. This volume documents subroutines with which users can create, destroy, and post and receive notices on their event synchronizers. It also describes subroutines that associate several event synchronizers into an event group.
- The Timers facility makes time-dependent process synchronization possible. This volume describes subroutines with which users can create, destroy, set and reset timers that post notices on event synchronizers at a specified time or interval.
- The ISC facility makes it possible for processes that are running concurrently to exchange messages. This volume describes subroutines for establishing a message session and sending and receiving messages between two processes. These processes may be running on the same

system or on two different systems connected by PRIMENET^{TM}. Message exchange is coordinated by using event synchronizers.

Specifics of This Volume

Volume II of the *Subroutines Reference* series presents detailed descriptions of system search rule subroutines and subroutines used in manipulating file system entities. It also describes subroutines related to EPF manipulation and the command environment.

Chapters 2, 3, and 4 describe three groups of file system subroutines: those that control access to objects, that attach to file directories, and that operate on (creating, using, and deleting) the objects themselves.

Chapter 5 describes subroutines that deal with the initialization, execution, and maintenance of executable program format (EPF) files, and the management of dynamic storage space required for their execution.

Chapter 6 describes a group of subroutines that enable user programs to take advantage of some of the functions built into the command environment: determining the command environment breadth and depth, setting and retrieving local and global variables, parsing command lines, and related operations. Some of these subroutines are particularly useful when used in routines that are called by CPL programs.

Chapter 7 describes system search rule subroutines that enable users to read and modify the sequential search lists that PRIMOS uses to locate file system objects.

The appendices provide information about obsolete subroutines, data type equivalents, and argument parsing by the CL$PIX subroutine.

Three indexes enable the reader to find information quickly.

- The Index of Subroutines by Function, a list of all subroutines in the five-volume series, grouped by the general types of function that they perform. Use this index to find out which subroutines perform a particular function, then use the Index of Subroutines by Name to locate the desired subroutine.
- The Index of Subroutines by Name, an alphabetical list of all subroutines in the five-volume series. It lists the volume, chapter, and page number of the reference material for each subroutine.
- The Volume Index, a list of the topics treated in this volume. Use this index to find out where in this volume a particular topic, process, or term is described.

Suggested References

The other volumes of the *Subroutines Reference* document set are the following:

- *Subroutines Reference I: Using Subroutines* (DOC10080–2LA) and its update for Rev. 23.0 (UPD10080–21A)
- *Subroutines Reference III: Operating System* (DOC10082–2LA)
- *Subroutines Reference IV: Libraries and I/O* (DOC10083–2LA)
- *Subroutines Reference V: Event Synchronization* (DOC10213–1LA) and its update for Rev. 23.0 (UPD10213–11A)

The five volumes of the *Subroutines Reference* and their current updates can be ordered as a set using DCP10159.

The *PRIMOS User's Guide* (DOC4130-5LA) 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. New information for Rev. 23.0 can be found in the *PRIMOS User's Release Document* (DOC10316–1PA).

Also available for Rev. 23.0 is the *Rev. 23.0 Software Release Document* (DOC10001–7PA). This contains information primarily of interest to System Administrators and operators.

The *Programmer's Guide to BIND and EPFs* (DOC8691-1LA) and its updates for Rev. 22.0 (UPD8691-11A) and Rev. 23.0 (UPD8691–12A) show application programmers how to use the executable program format environment.

The *Advanced Programmer's Guides,* the companions to the *Subroutines Reference* series, consist of four volumes:

- *Advanced Programmer's Guide I: BIND and EPFs* (DOC10055-2LA)
- *Advanced Programmer's Guide II: File System* (DOC10056-3LA)
- *Advanced Programmer's Guide III: Command Environment* (DOC10057-2LA)
- *Advanced Programmer's Guide: Appendices and Master Index* (DOC10066-4LA)

These volumes provide strategies for the use of subroutines by system programmers and application programmers. They provide the most complete information on the use of EPFs, of file system subroutines, and of command environments. The *Appendices and Master Index* volume contains an annotated listing of all PRIMOS standard error codes, as well as an index to the entire *Advanced Programmer's Guide* documents set.

The following related Prime publications are also available:

- *Operator's Guide to System Commands* (DOC9304-5LA)
- *System Administrator's Guide, Volume I: System Configuration* (DOC10131-3LA)
- *System Administrator's Guide, Volume II: Communication Lines and Controllers* (DOC10132-2LA), updated by RLN10132–21A.
- *System Administrator's Guide, Volume III: System Access and Security* (DOC10133-3LA)
- *System Architecture Reference Guide* (DOC9473-2LA)

For a complete list of available Prime documentation, consult the *Guide to Prime User Documents.*

Prime Documentation Conventions

Subroutine descriptions use the conventions shown below. Examples illustrate use of these conventions.

Convention	*Explanation*	*Example*
Uppercase	In subroutine descriptions, words in uppercase indicate actual names of commands, options, statements, data types, and keywords.	**FIXED BIN**
Italic	In subroutine descriptions, words in italic indicate variables for which you must substitute a suitable value.	*key,filename*
Abbreviations	If a subroutine has an abbreviation, the abbreviation is placed immediately below the full form.	**TMR$GTIM** **TMR$TM**
Bold	In the Usage section, bold indicates the DECLARE and CALL statements of the subroutine.	**CALL DATE$A (*date*)**
Bold italic	In the Usage section, bold italic indicates variable parameters of the subroutine.	***date***
Monospace	Words and characters in monospace indicate system output, for example, error messages, prompts, examples, and text in screens.	`FILE NOT FOUND`
Underscore (in examples)	In examples, user input is underscored, but system prompts and output are not.	`OK, CBL ROTATE`
Brackets	In DECLARE and CALL statements, brackets indicate an optional parameter or argument.	**[RETURNS(FIXED BIN(31)]**
Hyphen	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	**SPOOL -LIST**
Subscript	A subscript after a number indicates that the number is not in base 10. For example, the subscript 8 is used for octal numbers.	200_8
Parentheses	In CALL statements, parentheses must be entered exactly as shown.	**CALL TIMDAT (*array, n*)**

Overview of Subroutines

1

A **subroutine** is a module of code that can 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 can 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.

This chapter summarizes the calling conventions for Prime subroutines and explains the format of the subroutine descriptions. It assumes that readers know a high–level language or PMA (Prime Macro Assembler), and that they are familiar with the concept of external subroutines. For more information on calling subroutines from Prime languages, see the chapter on your particular language in Volume I.

Functions and Subroutines

In this guide, a **function** is a call that returns a value. You call a function by using it in an expression; the function's returned value can then be assigned to a variable or used in other operations within the expression. Here, the value returned by TNCHK$ is assigned to the variable VALUE1:

```
VALUE1 = TNCHK$(arg1, arg2);
```

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

```
CALL AC$SET(arg1, arg2, arg3, arg4);
```

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

Subroutine Descriptions

In this guide, each subroutine description contains the following sections (see Figure 1–1):

- *Usage.* The format of a subroutine declaration and a subroutine call, using PL/I language elements. For further information, see the section Subroutine Usage below.
- *Parameters.* Information about the arguments the subroutine expects and the values it returns. For further information, see the section Subroutine Parameters later in this chapter.
- *Discussion.* Additional information about the subroutine and examples of its use.
- *Loading and Linking Information.* Information about what libraries must be loaded during the loading and linking process. For more information, see Satisfying the References at Load Time later in this chapter.

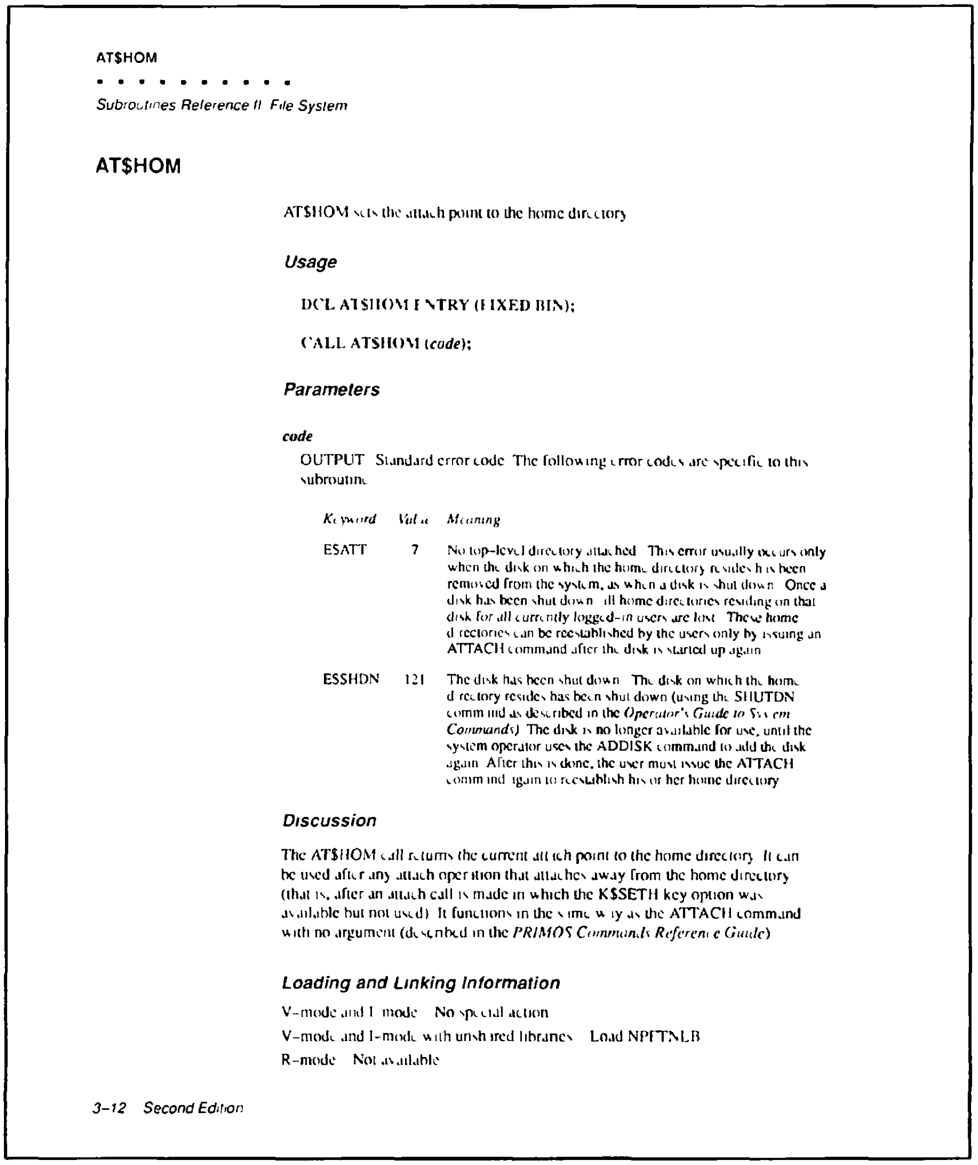
Subroutine Usage

The Usage section of each subroutine description includes two items of information:

- How to declare the subroutine in a program
- How to invoke it in a program

The notation used is that of the PL/I language. If you do not know PL/I, the explanation of the relevant PL/I syntax and data types in this section and the Subroutine Parameters section should enable you to call these subroutines from other languages.

Not all languages require that a subroutine be declared, but the Usage section should always be referred to for information on data types.



AT$HOM

Subroutines Reference II File System

AT$HOM

AT$HOM sets the attach point to the home directory

Usage

```
DCL AT$HOM ENTRY (FIXED BIN);

CALL AT$HOM (code);
```

Parameters

code

OUTPUT Standard error code The following error codes are specific to this subroutine

Keyword	*Value*	*Meaning*
ESATT	7	No top-level directory attached This error usually occurs only when the disk on which the home directory resides has been removed from the system, as when a disk is shut down Once a disk has been shut down all home directories residing on that disk for all currently logged-in users are lost These home directories can be reestablished by the users only by issuing an ATTACH command after the disk is started up again
ESSHDN	121	The disk has been shut down The disk on which the home directory resides has been shut down (using the SHUTDN command as described in the *Operator's Guide to System Commands)* The disk is no longer available for use, until the system operator uses the ADDISK command to add the disk again After this is done, the user must issue the ATTACH command again to reestablish his or her home directory

Discussion

The AT$HOM call returns the current attach point to the home directory It can be used after any attach operation that attaches away from the home directory (that is, after an attach call is made in which the K$SETH key option was available but not used) It functions in the same way as the ATTACH command with no argument (described in the *PRIMOS Commands Reference Guide*)

Loading and Linking Information

V-mode and I mode No special action

V-mode and I-mode with unshared libraries Load NPFTNLB

R-mode Not available

3-12 Second Edition

101.01.D10081.2LA

Figure 1-1. A Subroutines Description

Subroutine Declarations

The following example shows a subroutine declaration:

```
DCL AC$SET ENTRY (FIXED BIN, CHAR(128)VAR, PTR,
                  FIXED BIN);
```

DCL is the short form of DECLARE. The DECLARE statement is used to declare all data types, including those involved in subroutines and functions. AC$SET is the subroutine name. ENTRY specifies that the item being declared is an entrypoint in a subprogram external to the program from which it is called.

The items in parentheses are the parameters of the subroutine. The parameters indicate the data types required for each argument of the subroutine.

Subroutine Calls

The following example shows a call to the subroutine declared above:

```
CALL AC$SET (key, name, acl_ptr, code);
```

PL/I does not distinguish between uppercase and lowercase characters. In the Usage section of a subroutine description, lowercase letters indicate the items that must be supplied by the user, both arguments (actual parameters, as opposed to formal parameters) and data items. These are described more fully in the Parameters section. Uppercase letters indicate items that must be copied exactly as shown.

The CALL statement above invokes the subroutine AC$SET. The arguments in parentheses correspond to the parameters in the subroutine declaration. The variables or constants used as arguments in a call to the subroutine must match the data types of the parameters in the declaration. Here, the variable *name* must be a character string, while *key* and *code* must be integers. A subroutine that has no parameters is invoked simply by giving the CALL keyword and the name of the subroutine:

```
CALL TONL;
```

Function Declarations

The following example shows a function declaration:

```
DCL ISACL$ ENTRY (FIXED BIN, FIXED BIN) RETURNS (BIT(1));
```

The only difference between a function declaration and a subroutine declaration is at the end of the DECLARE statement. The function declaration contains the keyword RETURNS, followed by a **returns descriptor** specifying the data type of the value returned by the function. In this case, it is a logical or Boolean value — one that equates to TRUE or FALSE.

Function Calls

A function is invoked when its name is used as an expression on the right side of an assignment statement. The following example shows an invocation of the function declared above:

```
is_acl_dir = ISACL$ (unit, code);
```

The equal sign (=) is the assignment operator. *is_acl_dir* is a logical (Boolean) variable that is assigned the value returned by the call to ISACL$. *unit* and *code* represent integer values.

Functions Without Parameters

A function that takes no parameters is invoked with an empty argument list. The DATE$ subroutine is declared as follows:

```
DCL DATE$ ENTRY RETURNS(FIXED BIN(31));
```

Its invocation looks like this:

```
date_word = DATE$( );
```

Note Functions that take no arguments cannot be called from FTN programs; they can, however, be called from F77 programs.

Subroutine Parameters

Subroutines usually expect one or more arguments from the calling program. These arguments must be of the data type specified in the DECLARE statement. Volume I discusses how to translate the data types indicated by the PL/I declarations into other Prime languages. A chart summarizing data type equivalents for all Prime languages is in Appendix B of this volume.

You must provide the number of arguments expected by the subroutine, in the order in which they are expected. 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 correctly. A small number of subroutines, such as IOA$, accept varying numbers of arguments.

The Usage section of a subroutine description gives the data types of the parameters. The Parameters section explains what information these parameters contain and what they are used for. Each parameter description in this section begins with a word in uppercase that indicates whether the parameter is used for input or output:

- INPUT means that the parameter is used only for input, and that its value is not changed by the subroutine.
- OPTIONAL INPUT refers to an input parameter that may be omitted. See the section Optional Parameters later in this chapter.
- OUTPUT means that the parameter is used only for output. You do not have to initialize it before you call the subroutine.
- OPTIONAL OUTPUT refers to an output parameter that may be omitted. See the section Optional Parameters later in this chapter.
- INPUT/OUTPUT means that the parameter is used for both input and output. The argument you pass to it may be changed by the subroutine.
- INPUT –> OUTPUT refers to a situation in which
    - The parameter, an input parameter, is a pointer.
    - The data item to which the pointer points is not a parameter of the subroutine, but it is changed by the subroutine.
- RETURNED VALUE is the value returned by a function. (It is not, strictly speaking, a parameter.)
- OPTIONAL RETURNED VALUE is the value returned by a subroutine that can be called either as a function or as a procedure. See the section Optional Returned Values later in this chapter.

Parameters and Returned–value Data Types

A PL/I parameter specification consists simply of a list of the data types of the parameters. The data types you will encounter, both in the parameter list and in the RETURNS part of a function declaration, are the following:

CHAR(*n*)

Also specified as CHARACTER(*n*), CHARACTER(*n*) NONVARYING. Specifies a character string or array of length *n*. A CHAR(*n*) string is stored as a byte–aligned string, one character per byte. (A **byte** is 8 bits.)

CHAR(*)

Also CHARACTER(*), CHARACTER(*) NONVARYING. Specifies a character string or array whose length is unknown at the time of declaration. A CHAR(*) string is stored as a byte–aligned string, one character per byte.

CHAR(*n*) VAR

Also CHARACTER(n) VARYING. Specifies a character string or array whose length can be a maximum of *n* characters. The first two bytes (one halfword) of storage for a CHAR(n) VAR string contain an integer that specifies the string length; these are followed by the string, one character per byte.

CHAR(*) VAR

Also CHARACTER(*) VARYING. Specifies a character string or array whose length is unknown at the time of declaration. The first two bytes (one halfword) of storage for a CHAR(*) VAR string contain an integer that specifies the string length; these are followed by the string, one character per byte.

FIXED BIN

Also FIXED BINARY, BIN, FIXED BIN(15). Specifies a 16–bit (halfword) signed integer.

FIXED BIN(31)

Specifies a 32–bit signed integer.

(*n*) FIXED BIN

An integer array of *n* elements. See below for more information about arrays.

FLOAT BIN

Also FLOAT BIN(23), FLOAT. Specifies a 32–bit (one–word) floating–point number.

FLOAT BIN(47)

Specifies a 64–bit (double–word) floating–point number.

BIT(1)

Specifies a logical (Boolean) value. A bit value of 1 means TRUE; a value of 0 means FALSE.

BIT(*n*)

Specifies a bit string of length *n*. BIT(*n*) ALIGNED means that the bit string is to be aligned on a halfword boundary.

POINTER

Also PTR. Specifies a POINTER data type. A pointer is usually stored in three halfwords (48 bits). If the pointer will point only to haifword–aligned data, it may occupy two halfwords (32 bits). The item to which the pointer points is declared with the BASED attribute (for instance, BASED FIXED BIN).

POINTER OPTIONS (SHORT)
Same as POINTER except that it always occupies only two halfwords and can only point to halfword–aligned data.

Note When used as a parameter, POINTER can generally be used interchangeably with POINTER OPTIONS (SHORT).

When used as a returned function value, POINTER OPTIONS (SHORT) can be used in any high–level language except Pascal or 64V mode C, which require returned pointers to be three halfwords; in these cases, POINTER must be used. C in 32IX mode accepts only halfword–aligned, two–halfword pointers, and therefore requires the use of POINTER OPTIONS (SHORT).

Sometimes an argument is defined as an array or a structure. An array declaration looks like this:

```
DCL ITEMS(10) FIXED BIN;
```

Here, ITEMS is a 10–element array of integers. The keywords FIXED BIN, however, can be replaced by any data type. In PL/I, by default, arrays are indexed starting with the subscript 1; the first integer in this array is ITEMS(1).

An array with a starting subscript other than 1 is declared with a range specification:

```
DCL WORD(0:1023) BASED FIXED BIN;
```

WORD is an array indexed from 0 through 1023, and its elements are referenced by POINTER variables.

A structure is equivalent to a record in COBOL or Pascal. A structure declaration looks like this:

```
DCL 1 fs_date,
      2 year BIT(7),
      2 month BIT(4),
      2 day BIT(5),
      2 quadseconds FIXED BIN(15);
```

The numbers 1 and 2 indicate the relative level numbers of the items in the structure. The name of the structure itself is always declared at level 1. The level number is followed by the name of the data item and its data type. In this example, the structure occupies a total of 32 bits. (Remember that a FIXED BIN(15) value occupies 16 bits of storage.)

Since no names are given to data items in parameter lists, the array declared above as ITEMS would be declared simply as (10) FIXED BIN. Similarly, the structure FS_DATE would be listed as

```
(..., 1, 2 BIT(7), 2 BIT(4), 2 BIT(5), 2 FIXED BIN(15),
...)
```

Optional Parameters

On Prime computers, some subroutines and functions are designed so that one or more of their parameters, input or output, can be omitted. Candidates for omission are always the last *n* parameters. Thus, if a subroutine has a full complement of three parameters, it may be designed so that the last one or the last two can be omitted; the subroutine cannot be designed so that only the second parameter can be omitted. The first parameter can never be omitted.

In the Usage section of a subroutine description, any optional parameters are enclosed in square brackets, as in the following declaration and CALL statement:

```
DCL CNAM$$ ENTRY (CHAR(32), FIXED BIN, CHAR(32),
                  FIXED BIN, FIXED BIN [, FIXED BIN]);

CALL CNAM$$ (oldnam, oldlen, newnam, newlen, code
                  [, ok_open]);
```

In some cases, parameters can be omitted because they are not needed under the circumstances of the particular call. In other cases, when the parameter is of type INPUT, the subroutine will detect the missing parameter and will assume some value for it. For example, C1IN$, described in Volume III, can be called with one, two or three arguments:

```
CALL C1IN$ (char);
CALL C1IN$ (char, echo_flag);
CALL C1IN$ (char, echo_flag, term_flag);
```

If *echo_flag* is missing, the subroutine acts as if it had been supplied with a value of TRUE. If *term_flag* is missing, the subroutine acts as if it had been supplied with a value of FALSE.

In still other cases, the subroutine changes its behavior depending on the presence of the parameter. For example, the subroutine CH$FX1 (described in Volume III) uses its third argument to return an error code. If the code argument is omitted and an error occurs, the routine signals a condition instead.

If a parameter can be omitted, it is described as OPTIONAL INPUT or OPTIONAL OUTPUT in the routine description. Most of the routines described in the *Subroutines Reference* have no optional parameters.

Optional Returned Values

In the architecture of Prime computers, a subroutine that was designed as a function can be called as a subroutine using the CALL statement. Frequently this makes no sense. The statement

```
CALL SIN(45);
```

does nothing useful; the value that the SIN function returns is lost. But, with functions that change some of their parameters as well as return a value, the returned value can be useful in some contexts and not of interest in other contexts. Consider the function CL$GET, described in Volume III. It reads a line from the user terminal and, in addition, returns a flag that indicates whether a command input file is active. Most programs do not need to know whether a command input file is active. They would call CL$GET as a subroutine:

```
CALL CL$GET (BUFFER, 80, CODE);
```

A program that was interested in command input files, however, would call CL$GET as a function:

```
COMISW = CL$GET (BUFFER, 80, CODE);
```

Note In PL/I and Pascal, a given subroutine cannot be used both as a subroutine and as a function within a single source module.

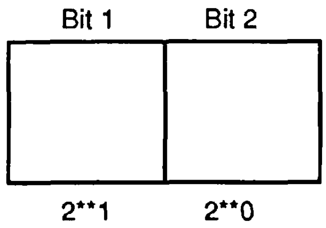
The Usage section of the subroutine descriptions gives both the function invocation and the subroutine invocation for subroutines that are likely to be called in both ways.

In the Parameters section, a routine that is designed as a function has its returned value described as RETURNED VALUE if it is considered the main purpose of the subroutine to return the value. If the function is likely to be called as a subroutine — that is, if returning the value is considered to be something that is needed only on some occasions — the returned value is described as OPTIONAL RETURNED VALUE.

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 the bit value '0'B or false, and on is equated to '1'B or true. (This is not the same as the FORTRAN values .FALSE. and .TRUE., which are the LOGICAL data type and are really integers.) When bits are stored as part of a group, however, the position of the bit gives it a numeric value as well as the bit value '1'B or '0'B. Its position equates it to a power of 2. Consider an argument that contains only two bits, represented in Figure 1–2.



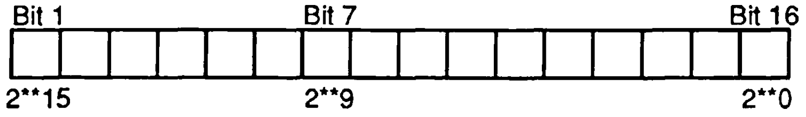
101.02.D10081.2LA

Figure 1–2. Values of Bit Positions — Two Bits

The *low–order* bit is in the position of 2 to the 0 power, and its value, if ON, is 1. The *high–order* bit is in the position of 2 to the first power, and its value, if ON, is 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. You can pass an integer as an argument that is declared as BIT(n) ALIGNED. The subroutine interprets the integer as a bit string. For example, if you want to set the sixteenth and the seventh bits, compute 2 to the 0 power plus 2 to the ninth power, which amounts to 1 plus 512, or 513. Figure 1–3 illustrates values of bit positions in a 16–bit argument.

If an argument is declared as BIT(1) or BIT(1) ALIGNED, the bit passed is the *most* significant (leftmost) bit of the memory location referred to.



101.03.D10081.2LA

Figure 1–3. Values of Bits in a 16–bit Argument

Key Names as Arguments

In calls to many subroutines, data names known as keys can be used to represent numeric arguments. The subroutine description explains which key to use.

Key Names as Arguments

In calls to many subroutines, data names known as keys can be used to represent numeric arguments. The subroutine description explains which key to use. Numeric values are associated with these keys in the SYSCOM directory. The keys in SYSCOM are listed in Volume I.

Keys are of the form *x$yyyy*, where *x* is either K or A and *yyyy* is any combination of letters. Keys that begin with *K* concern the file system; those that begin with *A* concern applications library routines. Examples are:

```
K$CURR
A$DEC
```

For example, in the subroutine call

```
CALL GPATH$ (K$UNIT.....other arguments...);
```

the key K$UNIT stands for a numeric constant value expected by the subroutine. If a subroutine expects key arguments, the description of that subroutine explains which keys to use in which circumstances.

Each language has its own files of keys. The chapters on individual languages in Volume I explain how to insert these files into your program. Key files have the pathnames

```
SYSCOM>KEYS.INS.language
```

for K$*yyyy* keys, and

```
SYSCOM>A$KEYS.INS.language
```

for A$*yyyy* keys, where *language* is the suffix for that language. For more information about keys, see Volume I.

Standard Error Codes

Many subroutines include as an argument a standard error code, which is similar to a key. The error code corresponds to an error message that the subroutine can return to indicate that the call to the subroutine succeeded or failed, or to report some other condition worth noting.

Standard error codes are of the form *E$xxxx*, where *xxxx* is any combination of letters. For example, the error code `E$DVIU` corresponds to the error message `The device is in use..`

The standard error codes are defined in the SYSCOM directory. Like a key file, the error code file for a particular language must be inserted in the program that calls the subroutine. Each error code file has the pathname

```
SYSCOM>ERRD.INS.language
```

where *language* is the suffix for that language. Volume I contains a listing of the standard error codes and the messages to which they correspond. For explanations of the standard error codes, see the *Advanced Programmer's Guide: Appendices and Master Index.*

Libraries and Addressing Modes

The *Subroutines Reference* is organized to give a systematic description of subroutine libraries — sets of routines, all broadly dealing with the same subject, grouped into one file. There is a separate library for each of these subjects.

Prime computers offer several addressing modes to provide source–level compatibility among several machine models. To maintain this compatibility, a given subroutine library normally exists in three general versions: V–mode, V–mode (Unshared), and R–mode. A discussion of shared and unshared libraries appears in Volume I of *Subroutines Reference.* For a description of addressing modes, see the *System Architecture Reference Guide.*

Programs compiled in either V–mode or I–mode can use either V–mode or I–mode libraries (V–mode libraries supplied by Prime serve both V–mode and I–mode programs). Programs written in R–mode *must* use the R–mode version of the library.

Loading and Linking Information

Every subroutine description contains a section entitled Loading and Linking Information, which describes what, if any, action to take to permit linking to the subroutine from programs in each of the compilation modes.

In these sections, some subroutines are designated as not available in one or more versions (most often the R–mode version). If a subroutine is not available in a given mode, it means that that subroutine cannot be called from a program written and compiled in that mode. For example, programs intended to manipulate EPFs using the EPF subroutines cannot be linked and executed in R–mode, since there are no R–mode versions of these subroutines. Such programs must be written, compiled, and linked in V–mode or I–mode.

Satisfying the References at Load Time

When subroutines are called by a program, the references must be satisfied when the compiled binaries are linked together with BIND, SEG, or LOAD (the R–mode loader).

This is accomplished by loading a binary library supplied by Prime using the LI (for Library) command. The Loading and Linking Information section under each subroutine description provides the information for up to three loading choices:

- V–mode or I–mode, with shared code. This is the preferred method, as it allows many users of a system to share the same copy of code.
- V–mode or I–mode with unshared code.
- R–mode.

For most subroutines described in this volume, only the V–mode or I–mode subroutines with unshared code require a special library. Both the shared version and the R–mode version (when available) require "no special action." This means that the LI[brary] command with no arguments, which normally ends a loading sequence, satisfies the references.

Getting the Subroutines at Runtime

When a subroutine is available to be shared between users, PRIMOS postpones finding the code until runtime. (Other subroutines have their code so linked with the program that they are called "unshared" routines.) The program linked to shared subroutine code contains only the name of the subroutine, and at runtime PRIMOS replaces the name with the actual location of the shared code, thus completing the connection. For the connection to happen, the code must be in one of three places: in PRIMOS itself, in an EPF library, or in a static–mode library. Furthermore, the user's ENTRY$ search list must contain a pathname to the library that holds the code, unless the subroutine is located in PRIMOS.

If the Loading and Linking Information section indicates "no special action" for loading a subroutine library, then the code for this subroutine is either in PRIMOS itself or in one of the two EPF libraries suplied by Prime, SYSTEM_LIBRARY.RUN or PRIMOS_LIBRARY.RUN. The pathnames to these libraries must be in the system search rules.

Because many of the subroutines described in this guide provide PRIMOS services, there is no way of providing them as unshared code, since PRIMOS by definition is shared. Even if you call these subroutines from programs that are loaded with unshared libraries, what is executed by these calls is shared code.

For a further description of libraries and related terminology, see *Subroutines Reference I: Using Subroutines.*

Access Control

2

Access control refers to the protection that PRIMOS and the user can specify for a file system object to prevent unauthorized access to it. Protection is defined by use of a list called an **access control list**, or **ACL**.

This chapter describes a set of system subroutines that can be used to manipulate the access control lists of file system objects.

Subroutines are provided to set, modify, and delete ACLs on most types of objects: access categories, user file directories, segment directories, and files. ACLs of master file directories (MFDs) can be manipulated only by a System Administrator or by a user working at the terminal designated as the supervisor terminal (User 1).

Several subroutines can be used to obtain access control information, while others can manipulate the older password–protected directories and files.

User programs can also use the ACL mechanism to control user access to resources other than files.

Detailed information on the use of ACLs can be found in the *PRIMOS User's Guide* and in the *Advanced Programmer's Guide II: File System.*

The following subroutines, their declarations, and their calling sequences are described in this chapter:

AC$CAT	Add an object's name to an access category.
AC$CHG	Modify an existing ACL on an object.
AC$DFT	Set an object's ACL to that of its parent directory.
AC$LIK	Set an object's ACL like that of another object.
AC$LST	Obtain the contents of an object's ACL.
AC$RVT	Convert an object from ACL protection to password protection.
AC$SET	Set a specific ACL on an object.
CALAC$	Determine whether an object is accessible for a given action.
CAT$DL	Delete an access category.
GETID$	Obtain the user ID and the groups to which it belongs.

GPAS$$	Obtain the passwords of a subdirectory of the current directory.
ISACL$	Determine whether an object is ACL–protected.
PA$DEL	Remove an object's priority access.
PA$LST	Obtain the contents of an object's priority ACL.
PA$SET	Set priority access on an object.
SPAS$$	Set the owner and nonowner passwords on an object.

AC$CAT

AC$CAT adds an object's name to an access category.

Usage

DCL AC$CAT ENTRY (CHAR(128)VAR, CHAR(32)VAR, FIXED BIN);

CALL AC$CAT (*name, category_name, code*);

Parameters

name

INPUT. Pathname or objectname of the object to be protected.

category_name

INPUT. Name of the category to which *object_path* is to be added.

code

OUTPUT. Standard error code.

Discussion

An access category provides protection to any number of objects without using the disk space that would be required to place a specific ACL on each of the objects. Since an access category uses about the same disk space as two average ACLs, whenever more than two objects require the same protection, the user should consider using an access category.

The object named in *name* must exist and must be a file, a file directory, or a segment directory. If the object is in the current directory, *name* can be a simple objectname.

The access category must exist in the same directory as the object. If the object is password–protected and its parent is an ACL directory, the object is converted to ACL protection.

Protect and List access are required on the parent directory if the object is a file; if it is a directory or an access category, Protect access is required on the object itself. If the object is a password directory and Protect access is not available on its parent, Owner access is required on the object. Use access is required for each intermediate subdirectory in the path.

To create an access category and to set specific ACLs, refer to the AC$SET subroutine, described later in this chapter.

For more information on the use of access categories, refer to the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$CHG

AC$CHG modifies an existing ACL on an object.

Usage

DCL AC$CHG ENTRY (CHAR(128)VAR, PTR, FIXED BIN);

CALL AC$CHG (*name, acl_ptr, code*);

Parameters

name

INPUT. Pathname or objectname of the object whose ACL is to be modified.

acl_ptr

INPUT. Pointer to the ACL structure (the structure declaration is described with AC$LST, later in this chapter).

code

OUTPUT. Standard error code.

Discussion

AC$CHG updates an existing ACL with new data. It performs the same function as the EDIT_ACCESS (EDAC) command described in the *PRIMOS Commands Reference Guide.* The object whose access is to be changed must be an existing access category or a specifically protected object. If it is not, an error is returned.

If the object whose ACL is to be changed is in the current directory, *name* can be a simple objectname.

The user specifies the changes to be made to the ACL by means of an ACL structure in the program, formatted as described under the AC$LST subroutine, later in this chapter. Each entry must have a user ID part, and may or may not have an access part. As in the EDAC command, if the access half of the user ID/access pair in the structure is null, the entry having this user ID in the ACL is removed from the ACL. If the user ID in the structure already exists in the ACL, this user's access is changed to that specified in the structure; if the user ID does not exist in the ACL, the user ID and its accompanying access half are added to the ACL.

Protect and List access are required on the parent directory if the object is a file, or on the object itself if it is a directory or access category. Use access is required for each intermediate subdirectory in the path. An attempt to use AC$CHG on an object with password protection returns an error.

For more information on manipulating access control lists, refer to the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$DFT

AC$DFT sets an object's ACL to that of its parent directory.

Usage

DCL AC$DFT ENTRY (CHAR(128)VAR, FIXED BIN);

CALL AC$DFT (*name, code*);

Parameters

name

INPUT. Pathname or objectname of the object whose protection is to be changed.

code

OUTPUT. Standard error code.

Discussion

The AC$DFT call sets the protection of the object named in *name* to that of the parent directory (which can itself default to that of a directory one or more levels higher). In the absence of any specific access control operations on a given object, the object always retains the default access it was given when it was created.

The object must exist when the AC$DFT call is made, and can be a file, a file directory, or a segment directory. If *name* is a password directory and its parent is an ACL directory, *name* is converted to an ACL directory. An attempt to use AC$DFT on an MFD is rejected.

AC$DFT requires Protect and List access for the parent of the object, or on the object itself if it is a directory. Use access is required at each intermediate subdirectory level. If the object is a password directory, Owner access is required if Protect access is not available on the parent.

For more information on manipulating access control lists, refer to the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$LIK

AC$LIK sets an object's ACL like that of another object.

Usage

```
DCL AC$LIK ENTRY (CHAR(128)VAR, CHAR(128)VAR,
                           FIXED BIN);

CALL AC$LIK (target_name, reference_name, code);
```

Parameters

target_name

INPUT. Pathname or objectname of the object to be protected.

reference_name

INPUT. Pathname or objectname of the object from which to take the ACL.

code

OUTPUT. Standard error code.

Discussion

Both *target_name* and *reference_name* must refer to existing file system objects. A new specific ACL is created for the target, giving it the same protection as 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 is converted to an ACL directory. The reverse is not true; that is, the AC$LIK call cannot be used to convert an ACL–protected object to a password–protected object.

target_name or *reference_name* (or both) can be a simple objectname if the object referred to is in the current directory.

AC$LIK requires Protect and List access to the target's parent, or Protect access to *target_name*. It also requires List access to the parent of *reference_name*.

For more information on manipulating access control lists, refer to the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$LST

AC$LST obtains the contents of an object's ACL.

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*);

Parameters

name

INPUT. Pathname or objectname of the object for which ACL contents are desired.

acl_ptr

INPUT –> OUTPUT. Pointer to user's ACL structure, described below.

max_entries

INPUT. Maximum number of entries that the user's defined structure can contain.

acl_name

OUTPUT. Name of the ACL protecting the object. The name is determined by the algorithm described in the Discussion section below.

acl_type

OUTPUT. Type of ACL protecting the object. Possible values are

0	Object protected by specific ACL.
1	Object protected by access category.
2	Default access provided by specific ACL for some parent directory.
3	Default access provided by an access category that is not in the directory that contains the object.
4	Object specified in *name* is an access category.

code

OUTPUT. Standard error code.

Discussion

AC$LST requires List access to the parent of the object.

If the object referred to in *name* is in the current directory, a simple objectname can be used in place of a pathname.

If *name* is null, the contents of the 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 having the following format:

```
dcl 1 acl,
      2 version FIXED BIN,   /* Must be 2. */
      2 entry_count FIXED BIN,
      2 entries(entry_count) CHAR(80) VAR;
```

version

INPUT. The calling program must specify the value 2 for version.

entry_count

OUTPUT. Number of entries returned to *entries*.

entries(entry_count)

OUTPUT. Each entry in *entries* is a string of the form <user_ID:access>. A valid entry might be HOLMES:LUR. The *user_ID* can also be a group name such as .PRIVATE_EYES. Group names start with a period.

For more information on manipulating access control lists, refer to the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$RVT

AC$RVT converts an object from ACL protection to password protection.

Usage

```
DCL AC$RVT ENTRY (FIXED BIN);

CALL AC$RVT (code);
```

Parameters

code

OUTPUT. Standard error code. Possible values are

Keyword	*Meaning*
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 converts the current directory to a password directory. The directory must not contain any access categories or ACL subdirectories; if it does, the call is rejected.

Protect access is required on the current directory. The SPAS$$ call can be used to set owner and nonowner passwords on the converted directory to other than their defaults of spaces and nulls, respectively.

AC$RVT is provided for compatibility with systems that still use password protection. The use of password protection is discouraged in new programming. Information on the conversion of password directories to ACL directories is given in the *PRIMOS User's Guide*.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

AC$SET

AC$SET sets a specific ACL on an object.

Usage

**DCL AC$SET ENTRY (FIXED BIN, CHAR(128)VAR, PTR,
FIXED BIN);**

CALL AC$SET (*key, name, acl_ptr, code*);

Parameters

key

INPUT. Indicates caller's intentions. Possible values are

Keyword	*Meaning*
0	Create a new ACL if one does not exist; replace it if it already exists.
K$CREA	Create a new ACL if one does not exist; return an error if one already exists.
K$REP	Replace the contents of an existing ACL; return an error if one does not exist.

name

INPUT. Pathname of the file system object to be protected.

acl_ptr

INPUT. Pointer to an ACL structure declared in the user program and formatted as for AC$LST, described earlier.

code

OUTPUT. Standard error code.

Discussion

The AC$SET call provides user programs with a method of creating and replacing the ACL of an access category, a file, a file directory, or a segment directory. If the object referred to in *name* is in the current directory, a simple objectname can be used in place of a pathname.

The structure in which the access control information is defined is declared in the user program in the format described for the AC$LST call earlier in this chapter. In the absence of an entry in the structure for the special user group $REST, the AC$SET call automatically provides a $REST:NONE entry in the resulting ACL.

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 K$CREA, an error is returned. Otherwise, the category's existing ACL is replaced with the new one pointed to by *acl_ptr*.
- The named object is a file, a file directory, or a segment directory. If the file is protected by a specific ACL and the key is K$CREA, an error is returned. Otherwise, a new specific ACL is created and the object is pointed to it. Any existing specific ACL is deleted. If the object is a password directory and its parent is an ACL directory, it is 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.

To add a file system object to an existing access category, refer to the AC$CAT subroutine, described earlier in this chapter.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

CALAC$

Determines whether an object is accessible for a given action.

Usage

```
DCL CALAC$ ENTRY (CHAR(128)VAR, PTR, CHAR(47)VAR,
                  CHAR(47)VAR, FIXED BIN)
                  RETURNS (BIT(1));
```

have_access = CALAC$ (*name, id_ptr, acc_needed, acc_gotten, code*);

Parameters

name

INPUT. Pathname of the file system object to check.

id_ptr

INPUT. Pointer to the user ID structure.

acc_needed

INPUT. A list of accesses required (ignored if object is password–protected).

acc_gotten

OUTPUT. The list of accesses available.

code

OUTPUT. Standard error code.

have_access

RETURNED VALUE. True if *acc_needed* is a subset of *acc_gotten*, or if the object is password–protected (in which case *acc_needed* is ignored).

Discussion

The user ID structure pointed to by *id_ptr* is the same as that for GETID$, described later in this chapter. 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 one or more of the letters P, D, A, L, U, R, and W, or the special modes ALL and NONE.

If the object referred to in *name* is in the current directory, a simple objectname can be used in place of a pathname. If *name* is null, the rights for the current directory are returned.

If CALAC$ determines that the object is password–protected, password rights are returned in *acc_gotten.* 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 are 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.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

CAT$DL

CAT$DL deletes an access category.

Usage

DCL CAT$DL ENTRY (CHAR(128)VAR, FIXED BIN);

CALL CAT$DL (*name, code*);

Parameters

name

INPUT. Pathname of the access category to be deleted.

code

OUTPUT. Standard error code.

Discussion

The object specified in *name* must exist and must be an access category. If it is in the current directory, a simple objectname can be used in place of a pathname.

When an access category is deleted, any objects that were protected by it revert to default access (the access of their parent directory).

A specific ACL cannot be explicitly deleted. It is deleted by PRIMOS when the object it protects is

- Deleted
- Put into an access category
- Given default protection

An access category that protects the MFD cannot be deleted.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

GETID$

Obtains the user ID and the groups to which it belongs.

Usage

DCL GETID$ ENTRY (PTR, FIXED BIN, FIXED BIN);

CALL GETID$ (*id_ptr, max_groups, code*);

Parameters

id_ptr

INPUT –> OUTPUT. Pointer to the *full_id* structure, described in the next section.

max_groups

INPUT. Maximum number of groups that the caller's *full_id* structure can contain.

code

OUTPUT. Standard error code. Possible values are

E$BPAR	*id_ptr* is null or *max_groups* is less than zero.
E$BVER	Invalid version number.

Discussion

The structure pointed to by *id_ptr* has the following format:

```
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 1 or 2 in Rev. 20.2.

user_id

The ID of the calling user.

group_count

Number of groups returned to the caller. This is always the lesser of the number specified in *max_groups* and the number of groups of which the user is a member. In Rev. 20.2, a user can be a member of up to 32 groups. If *max_groups* is 0, this field is not returned.

groups

The list of groups of which the user is a member.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

GPAS$$

GPAS$$ obtains the passwords of a subdirectory of the current directory.

Usage

DCL GPAS$$ ENTRY (CHAR(32), FIXED BIN, CHAR(6), CHAR(6),
FIXED BIN);

CALL GPAS$$ (*dirnam, namlen, opass, npass, code*);

Parameters

dirnam

INPUT. Name of the directory whose passwords are to be returned.

namlen

INPUT. Length in characters (1–32) of *dirnam.*

opass

OUTPUT. Owner password for *dirnam.*

npass

OUTPUT. Nonowner password for *dirnam.*

code

OUTPUT. Standard error code.

Discussion

GPAS$$ searches for *dirnam* in the current directory; therefore, only a simple objectname can be specified in *dirnam.*

GPAS$$ requires Protect access to the current directory.

The following example reads both passwords of SUBDIR:

```
dcl gpas$$ entry (char(32), fixed bin, char(6) char(6),
                  fixed bin);
dcl  mypass char(6);      /* owner password */
dcl  yourpass char(6);    /* nonowner password */
call gpas$$ ('subufd', 6, mypass, yourpass, code);
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

ISACL$

ISACL$ determines whether an object is ACL–protected.

Usage

DCL ISACL$ ENTRY (FIXED BIN, FIXED BIN) RETURNS (BIT(1));

is_acl_dir = ISACL$ (***unit, code***);

Parameters

unit

INPUT. File unit to check. *unit* is either a file unit number or one of the following:

–1	Current directory
–2	Home directory
–3	Initial directory

code

OUTPUT. Standard error code.

is_acl_dir

RETURNED VALUE. TRUE if directory specified in *unit* is an ACL directory; otherwise returns FALSE.

Discussion

For purposes of compatibility, ACL directories and password directories have the same type (as visible to users — internally they are different). Therefore, some means of distinguishing between the two is needed. ISACL$ is a function call that looks at the directory open on *unit* and returns TRUE if the directory is an ACL directory.

Information on ACL and password directories can be found in the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

PA$DEL

PA$DEL removes an object's priority access.

Usage

DCL PA$DEL ENTRY (CHAR(32)VAR, FIXED BIN);

CALL PA$DEL (*partition_name, code*);

Parameters

partition_name

INPUT. Name of the partition from which to remove a priority ACL.

code

OUTPUT. Standard error code.

Discussion

Use of the PA$DEL subroutine is restricted to User 1 (the supervisor terminal) and the System Administrator.

Refer to the PA$SET subroutine, later in this chapter, and to the *System Administrator's Guide, Volume III: System Access and Security* for a discussion of priority access and when and why it is used.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

PA$LST

Obtains the contents of an object's priority ACL.

Usage

```
DCL PA$LST ENTRY (CHAR(128)VAR, PTR, FIXED BIN,
                  FIXED BIN);

CALL PA$LST (name, acl_ptr, max_entries, code);
```

Parameters

name

INPUT. Pathname or objectname of any object on the partition whose priority ACL is to be read.

acl_ptr

INPUT –> OUTPUT. Pointer to ACL structure (described under AC$LST, earlier in this chapter).

max_entries

INPUT. Maximum number of entries the caller's structure can contain.

code

OUTPUT. Standard error code.

Discussion

The PA$LST call returns the same kind of information as the AC$LST call does; PA$LST, however, limits its returned information to that contained in a priority access control list previously created by a PA$SET call. The structure containing the returned information is declared in the user program in the same format as for the AC$LST call, described earlier in this chapter.

Unlike the PA$DEL and PA$SET calls, use of the PA$LST call is not restricted to User 1 or the System Administrator; it can be called by any user who satisfies access control requirements.

Normally, List access to the partition is required in order to determine the logical device number, and, through that number, to get the priority ACL. Since a priority ACL can be defined to disallow all access to a partition, PA$LST can be called with only a partition name (in angle brackets). In that case, it merely looks up the partition in the logical disk table and no access is required.

Refer to the PA$SET subroutine, later in this chapter, and to the *System Administrator's Guide, Volume III: System Access and Security* for a discussion of priority access and when and why it is used.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

PA$SET

PA$SET sets priority access on an object.

Usage

DCL PA$SET ENTRY (CHAR(32)VAR, PTR, FIXED BIN);

CALL PA$SET (*partition_name, acl_ptr, code*);

Parameters

partition_name
INPUT. Name of the partition to be protected.

acl_ptr
INPUT. Pointer to ACL structure.

code
OUTPUT. Standard error code.

Discussion

It is at times necessary for User 1 (the supervisor terminal) or the System Administrator to take exclusive control of a partition for the purpose of troubleshooting, taking system backups, or other procedures that cannot tolerate interference from other users. Under these circumstances, priority access can be set on the partition involved. **Priority access** does not disturb existing ACLs; it introduces, while it is in effect, a level of protection that takes precedence over an existing ACL. When this precedence is no longer required, priority access is removed using the PA$DEL call described earlier.

acl_ptr points to an ACL structure as described for the AC$LST subroutine earlier in this chapter. Any existing priority ACL on the specified partition is replaced by the new one. Unlike the action of the AC$SET subroutine, if no $REST entry is in the ACL passed to PA$SET, no $REST:NONE entry is supplied.

Refer to the *System Administrator's Guide, Volume III: System Access and Security* for more information on priority access and how to use it, and to the *PRIMOS User's Guide* for more information on access control.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

SPAS$$

SPAS$$ sets the owner and nonowner passwords on an object.

Usage

DCL SPAS$$ ENTRY (CHAR(6), CHAR(6), FIXED BIN);

CALL SPAS$$ (*owner_pw, nonowner_pw, code*);

Parameters

owner_pw
INPUT. Password to set as the owner password.

nonowner_pw
INPUT. Password to set as the nonowner password.

code
OUTPUT. Standard error code.

Discussion

SPAS$$ requires Owner rights to the current directory. 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 can contain any other characters including control characters.

Passwords that are intended to be accessed only through programs can have any bit pattern.

If the owner password supplied in the call is null, the owner password on the directory is set to spaces. If the nonowner password supplied in the call is null, the nonowner password on the directory is set to null (all 0 bits).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

3 Attaching

Attaching is the mechanism by which a user's process becomes connected to a file directory upon which (or subordinate to which) some operation is to be done. This is known as setting the current attach point.

Setting the current attach point always defines the user's **current** directory (sometimes known as the **cache** directory). In some cases the **home** directory can also be defined in the same call by the appropriate use of a *key* argument. Some routines temporarily alter the current attach point during their execution; they then reset the current attach point to be the same as the home attach point. See the *PRIMOS User's Guide* for an introductory discussion of attach points.

Note With the AT$, AT$ABS, AT$LDEV, and AT$OR subroutines, it is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value for these subroutines to ensure that your attach point after the completion of the subroutine is correct.

This chapter describes a set of system subroutines that can be used to set the current attach point to specified directories anywhere in that portion of the file hierarchy that is visible to the calling system. Programmers who need further information about these subroutines should see the *Advanced Programmer's Guide II: File System.*

With the introduction of the Rev. 23.0 file system and the concept of the common file system name space, it is important that you set up your search rules so that you can more quickly access those disks that you use most frequently. Refer to the *Advanced Programmer's Guide II: File System* for a detailed description of the Rev. 23.0 file system.

The following subroutines, their declarations, and their calling sequences are described in this chapter:

AT$	Set the attach point to a directory specified by pathname.
AT$ABS	Set the attach point to a specified top–level directory and partition.
AT$ANY	Set the attach point to a specified top–level directory on any partition.
AT$HOM	Set the attach point to the home directory.

AT$LDEV Set the attach point to a specified top–level directory on a partition identified by logical disk number.

AT$OR Set the attach point to the login directory.

AT$REL Set the attach point to a directory subordinate to the current directory.

AT$ROOT Set the attach point to the root directory.

ATCH$$ Set the attach point to a specified directory and, optionally, make it the home directory.

GTROB$ Determine whether current attach point exists on a robust partition.

AT$

AT$ sets the attach point to a directory specified by pathname.

Usage

DCL AT$ ENTRY (FIXED BIN, CHAR(128) VAR, FIXED BIN);

CALL AT$ (*key, name, code*);

Parameters

key

INPUT. Indicates whether the home as well as the current attach point should be set. Possible values are

K$SETC	Set current attach point only.
K$SETH	Set home and current attach points.

Note It is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value to ensure that your program returns to the desired attach point after the completion of the subroutine.

name

INPUT. Pathname or objectname of the directory to be attached to.

code

OUTPUT. Standard error code. Possible values are

E$BKEY	An invalid key value was passed.
E$ITRE	The treename was invalid.
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

The AT$ subroutine parses a pathname and passes the call to the AT$ABS, AT$ANY, AT$HOM, AT$ROOT, and AT$REL subroutines (described later in this chapter) to perform the actual attaching.

AT$ allows the user to do a pathname attach in one call. The subroutine to which the call is passed by AT$ depends on the form of the pathname. The several forms and their corresponding implementation are as follows:

Form	*Result*
<	Passed to AT$ROOT to attach to the root directory.
<*>	Passed to AT$ABS to attach to the current partition's MFD (the MFD containing the home directory in effect at the time of the AT$ call). The < in this special syntax does *not* indicate the root.
<dir>... *<dir*	Passed to AT$ROOT to attach to the specified root–level directory, followed by calls to AT$REL to attach to directories following the *<dir>* portion of the pathname.
*>...	Passed to AT$HOM to attach to the home directory, followed by calls to AT$REL to attach downward.
dir	Passed to AT$ANY to attach to a top–level directory, that is, a directory immediately subordinate to a partition's MFD.
dir>...	Passed to AT$ANY to attach to an absolute pathname, the first element being a top–level directory.
(null)	A null pathname has the same effect as using the AT$HOM call, described later in this chapter.

Note For many commands, such as COPY or SLIST, as well as for many subroutine calls, a simple objectname refers to an object in the *current* directory. When dealing with the AT$ subroutine, however, always keep in mind that a pathname whose first (or only) element is an objectname (is not an asterisk or a partition name enclosed in angle brackets) refers to a top–level directory called *objectname*, not a subdirectory in the current directory.

At Rev. 23.0, the concept of the **root directory** is introduced. To avoid any confusion regarding the hierarchy of directory pathnames, keep in mind that a **top–level directory** is any directory immediately subordinate to the MFD. The meaning of the term top–level has not changed in a functional sense. That is, the AT$ and AT$ANY subroutines function as they did before Rev. 23.0.

Be aware that with lower–mounted directories, they may no longer be considered top–level. See the discussion at AT$ABS. Refer also to the discussion on the

Rev. 23.0 common file system name space in the *Advanced Programmer's Guide II: File System.* If you have directories that are located in lower–mounted partitions, you may want to consider placing the mount–point pathname, plus the name of the lower–mounted partition, in the system search rules (ATTACH$).

Use access is required to each directory appearing in a pathname, including the MFD.

If *name* is a password directory with both an owner and a nonowner password, and the supplied password matches neither, two things happen: first, there is a five–second delay to discourage machine–aided breaking of passwords; second, the BAD_PASSWORD$ condition is signalled, but no error code is returned.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$ABS

AT$ABS sets the attach point to a specified top–level directory and partition.

Usage

```
DCL AT$ABS ENTRY (FIXED BIN, CHAR(32)VAR, CHAR(39)VAR,
                  FIXED BIN);
```

CALL AT$ABS (*key, part_name, dir_name, code*);

Parameters

key

INPUT. Specifies which attach points to change. Possible values are

Keyword	*Value*	*Meaning*
K$SETC	0	Set current attach point only.
K$SETH	1	Set current and home attach points.

Note It is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value to ensure that your program returns to the desired attach point after the completion of the subroutine.

part_name

INPUT. Name of the disk partition on which the directory resides.

dir_name

INPUT. Name of the directory to attach to. To specify a password, append it to *dir_name* with a single space separating *dir_name* and the password.

code

OUTPUT. Standard error code. If *code* is 0, the operation was successful. Otherwise, *code* is always positive. Error codes specific to this operation may have the following values.

E$BPAR	6	Bad parameter. The length of the directory name as passed by the calling program is a negative number or is greater than 39 (including an optional directory password).
E$NATT	7	No directory attached to. This error can occur only when the partition name is * and the partition on which the current directory resides is removed from the system, as when a disk is shut down. Use one of the subroutines described in this chapter to reestablish a current attach point.
E$FNTF	15	Not found. The specified partition does not exist, or the specified directory does not exist on that partition.
E$BNAM	17	Illegal name. The partition name must be between 0 and 32 characters in length. The directory name must also be between 0 and 32 characters in length (inclusive), optionally followed by a single space and a password from 1 to 6 characters long (inclusive).

Discussion

AT$ABS uses a partition name to specify the partition containing the directory to be attached to. To attach via a logical disk number, use AT$LDEV, described later in this chapter.

If *part_name* is null, logical device 0 (the command device) is assumed. If *part_name* is *, the partition containing the home directory at the time of the AT$ABS call is searched. If *dir_name* is null, the MFD is assumed.

A null partition name specifies logical disk 0 (the command device). The partition name *must* be the name of the root entry. At Rev. 23.0, the partition name is any valid directory name up to 32 characters long.

A partition name of * specifies the MFD mount point of the current attach point. If the current attach point is the root directory, * refers to the root directory itself.

Example: The following PL/I statement sets the home and current attach points to the directory named ORANGE on the partition named RHYMES:

```
  at$abs(k$seth,'RHYMES','ORANGE',code);
```

Before Rev. 23.0, AT$ABS allowed you to set the attach point to a specified top–level directory on a given partition. At Rev. 23.0, however, a disk partition may be logically mounted anywhere on the file system tree, not just directly below the root. If the desired directory is under a disk partition which is itself a lower–mounted directory, then AT$ABS will not locate the directory. In this case, use AT$ instead, as in the following statement.

```
at$(k$seth,'<RHYMES>ORANGE', code);
```

AT$ABS may not work as expected if you are using logical mounts. For information on logical mounts, refere to the *System Administrator's Guide, Volume I: System Configuration.*

You may specify the partition argument by using any of the following:

- The name of the partition
- The name of the partition on which the current directory resides
- The name of the partition corresponding to logical disk 0
- The name of the partition corresponding to a particular logical disk number

When your program calls AT$ABS, it provides

- A key that specifies whether the home attach point is to be set
- The identity of the directory's partition, in any of the forms listed above
- The name of the directory itself

The AT$ABS subroutine attempts to set the current attach point to the specified directory on the specified partition, and returns a code indicating whether the operation was successful. If the operation fails, no changes are made to the attach points. If the operation succeeds, the home attach point is also set to the current attach point if specified by the key.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$ANY

AT$ANY sets the attach point to a specified top–level directory on any partition.

Usage

DCL AT$ANY ENTRY (FIXED BIN, CHAR(39)VAR, FIXED BIN);

CALL AT$ANY (*key, dir_name, code*);

Parameters

key

INPUT. Specifies which attach points to change. Possible values are

K$SETC	Set only current attach point.
K$SETH	Set current and home attach points.

Note It is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value to ensure that your program returns to the desired attach point after the completion of the subroutine.

dir_name

INPUT. Name of the directory, including the password (if any), separated from the directory name by a space.

code

OUTPUT. Standard error code. The following error codes are specific to this subroutine:

Keyword	*Value*	*Meaning*
E$BPAR	6	Bad parameter. The length of the directory name as passed by the calling program is a negative number or is greater than 39 (including an optional directory password).
E$BNAM	17	Illegal name. The syntax of the directory name as supplied by the calling program is not correct. The directory name must be between 0 and 32 characters in length, optionally followed by a single space and a password. See the *PRIMOS User's Guide* for a description of the legal syntax for objectnames.

Keyword	*Value*	*Meaning*
E$NFAS	189	Top–level directory not found or inaccessible. The specified directory could not be found, or resides on a disk partition that cannot be accessed by the user.

Discussion

Before Rev. 23.0, AT$ANY was used to attach to a top–level directory on any disk partition. At Rev. 23.0, however, AT$ANY functions differently to accommodate the singly–rooted file system hierarchy and the Global Mount Table.

At Rev. 23.0, AT$ANY scans the entire root structure for the desired directory because a logical disk may be grafted at any point on the root hierarchy; the term **top–level** is any directory immediately subordinate to the MFD. The meaning of the term has not changed in a functional sense at Rev. 23.0. That is, the AT$ and AT$ANY subroutines function as they did before Rev. 23.0.

AT$ANY uses the ATTACH$ search rule to attach to the specified directory. Use the LIST_MOUNTS command or the NAM$L_GMT subroutine to determine the logical disk order for your system (or use the STATUS DISKS command if your system is not part of a common file system name space). For more information on the ATTACH$ search rule, see the *Advanced Programmer's Guide II: File System.*

The AT$ANY subroutine provides the following:

- A key that specifies whether the home attach point is to be set
- The name of the directory

The AT$ANY subroutine attempts to set the current attach point to the specified directory on the first partition it finds having such a directory. It returns a code indicating whether the operation was successful. If the operation fails, no changes are made to the attach points. If the operation succeeds, the home attach point is also set to the current attach point, if specified by the key.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$HOM

AT$HOM sets the attach point to the home directory.

Usage

DCL AT$HOM ENTRY (FIXED BIN);

CALL AT$HOM (*code*);

Parameters

code

OUTPUT. Standard error code. The following error codes are specific to this subroutine:

Keyword	*Value*	*Meaning*
E$ATT	7	No top–level directory attached. This error usually occurs only when the disk on which the home directory resides has been removed from the system, as when a disk is shut down. Once a disk has been shut down, all home directories residing on that disk for all currently logged–in users are lost. These home directories can be reestablished by the users only by issuing an ATTACH command after the disk is started up again.
E$SHDN	121	The disk has been shut down. The disk on which the home directory resides has been shut down (using the SHUTDN command as described in the *Operator's Guide to System Commands).* The disk is no longer available for use, until the system operator uses the ADDISK command to add the disk again. After this is done, the user must issue the ATTACH command again to reestablish his or her home directory.

Discussion

The AT$HOM call returns the current attach point to the home directory. It can be used after any attach operation that attaches away from the home directory (that is, after an attach call is made in which the K$SETH key option was available but not used). It functions in the same way as the ATTACH command with no argument (described in the *PRIMOS Commands Reference Guide*).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$LDEV

AT$LDEV sets the attach point to a specified top–level directory on a partition identified by logical disk number.

Usage

```
DCL AT$LDEV ENTRY (FIXED BIN, FIXED BIN, CHAR(39) VAR,
                        CHAR(32) VAR, FIXED BIN);

CALL AT$LDEV (key, ldev, dir_name, part_name, code);
```

Parameters

key

INPUT. Specifies which attach points to change. Possible values are

K$SETC	Set only current attach point.
K$SETH	Set current and home attach points.

Note It is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value to ensure that your program returns to the desired attach point after the completion of the subroutine.

ldev

INPUT. Logical device number of the partition on which to look for the top–level directory.

dir_name

INPUT. Name of the top–level directory to attach to, including the password (if any), separated from the directory name by a space. If null, the MFD is assumed.

part_name

OUTPUT. Name of the partition corresponding to *ldev*.

code

OUTPUT. Standard error code.

Discussion

The AT$LDEV subroutine provides an alternative way to attach to a top–level directory, using the logical disk number of the partition on which the directory resides rather than the partition name used with the AT$ABS call. AT$LDEV looks up the partition name corresponding to the supplied disk number, and passes this name, along with the rest of the arguments in the AT$LDEV call, to the AT$ABS subroutine through an internal call.

The *key* argument determines whether or not to set the attach point of the home directory, as well as the current directory, to the top–level directory named in *dir_name.*

The *ldev* argument must be between 0 and the highest logical disk number in the system's logical disk list. (Display the logical disk list by using the STATUS DISK command.)

If *dir_name* is a password directory and a password is included in the argument, the user is attached to the directory with owner or nonowner rights, depending on whether the owner password or the nonowner password was supplied. If the password is not included, or is neither the owner nor the nonowner password, the attachment is with nonowner rights. (The password, when supplied, is separated from the directory name by a space.)

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$OR

AT$OR sets the attach point to the login directory.

Usage

DCL AT$OR ENTRY (FIXED BIN, FIXED BIN);

CALL AT$OR (*key, code*);

Parameters

key

INPUT. Specifies which attach points to change. Possible values are

K$SETC	Set only current attach point.
K$SETH	Set current and home attach points.

Note It is possible to set either the current attach point or both the current and home attach points. Care should be taken to select the correct *key* value to ensure that your program returns to the desired attach point after the completion of the subroutine.

code

OUTPUT. Standard error code. The following error codes are specific to this subroutine:

Keyword	*Value*	*Meaning*
E$ATT	7	No top–level directory attached. This error usually occurs only when the disk on which the home directory resides has been removed from the system, as when a disk is shut down. Once a disk has been shut down, all home directories residing on that disk for all currently logged–in users are lost. These home directories can be reestablished by the users only by issuing an ATTACH command after the disk is started up again.

Keyword	*Value*	*Meaning*
E$SHDN	121	The disk has been shut down. The disk on which the home directory resides has been shut down (using the SHUTDN command as described in the *Operator's Guide to System Commands).* The disk is no longer available for use, until the system operator uses the ADDISK command to add the disk again. After this is done, the user must issue the ATTACH command again to reestablish his or her home directory.

Discussion

A user's process, when the user first logs in, is attached to the directory designated by the System Administrator as that user's *login*, or *origin*, directory. During a terminal session, the process will frequently attach to other directories (sometimes, perhaps, unbeknownst to the caller). The AT$OR call is used to reconnect the process to the origin directory; it functions in the same way as the ORIGIN command (described in the *PRIMOS Commands Reference Guide*).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$REL

AT$REL sets the attach point to a directory subordinate to the current directory.

Usage

DCL AT$REL ENTRY (FIXED BIN, CHAR(39)VAR, FIXED BIN);

CALL AT$REL (*key, dir_name, code*);

Parameters

key

INPUT. Specifies which attach points to change. Possible values are

K$SETC	Set only current attach point.
K$SETH	Set current and home attach points.

dir_name

INPUT. Name of the directory, including the password, if any, separated from the directory name by a space. *dir_name* must exist in, and be immediately subordinate to, the current directory.

code

OUTPUT. Standard error code.

Discussion

The AT$REL call enables the user program to attach to a subdirectory at the next level down from the current directory. AT$REL must be called once for each level the program needs to go down. Each call results in setting the current attach point (and optionally the home attach point) one level lower.

The AT$ subroutine, described earlier in this chapter, can be used to attach, through a single subroutine call, to a directory more than one level down from the current directory; use the AT$ call with the following pathname form:

```
*>dir_name_1>dir_name_2>...
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

AT$ROOT

AT$ROOT attaches to the root directory by means of the calling program.

Usage

DCL AT$ROOT ENTRY (FIXED BIN, FIXED BIN);

CALL AT$ROOT (*key, code*);

Parameters

key

INPUT. Indicates whether to set only the current attach point or to set both the current attach point and the home attach point. Possible values are

K$SETC	Set current attach point only.
K$SETH	Set both home and current attach points.

code

OUTPUT. Standard error code. Possible value is

E$BKEY	An invalid key was passed.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

ATCH$$

ATCH$$ sets the attach point to a specified directory and, optionally, makes it the home directory.

This subroutine is considered obsolete, and its use in new programming is discouraged. Use an appropriate AT$ call instead. Users maintaining existing programs that call ATCH$ can refer to Appendix A for a complete description of the subroutine.

GTROB$

GTROB$ determines whether a specified file exists on a robust partition or on a nonrobust partition.

Usage

DCL GTROB$ ENTRY (FIXED BIN(15), FIXED BIN(15))
RETURNS (BIT(1) ALIGNED);

robust_status = **GTROB$ (*funit, code*);**

Parameters

funit

INPUT. The file unit number of the given file.

code

OUTPUT. The status code. Possible values are

E$OK	Execution completed without error.
E$UNOP	File unit specified in *funit* is closed.
E$BUNT	A bad file unit number was specified in *funit*.

Network errors may be reported if the partition is remote and a network error occurs.

robust_status

RETURNED VALUE. TRUE if the specified file exists on a robust partition, or FALSE if it does not exist on a robust partition or if there is an error. GTROB$ sets the *robust_status* value to TRUE by setting its most significant bit to 1. It sets *robust_status* to FALSE by returning a value of 0.

Discussion

GTROB$ determines whether a given file, as specified by its file unit number, exists on a robust partition or on a nonrobust partition.

Loading and Linking Information

The dynamic link for GTROB$ is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

4 File and Directory Manipulation

This chapter describes the group of subroutines and functions used to perform various actions on file system objects after a user's process has met access control requirements and set its attach point to the appropriate place in the file system.

Subroutines are provided to perform the general categories of actions listed below:

- Creating and deleting file system objects
- Obtaining information about disks in use and their locations, and about directories, files, and their attributes
- Opening named files, file directories, and segment directories
- Opening numbered segment directory entries
- Reading, writing, positioning, and checking the existence of file system objects
- Extending and truncating CAM files; retrieving extent maps; setting CAM files' allocation size values
- Closing file system objects by name or file unit number
- Manipulating names, suffixes, attributes, read/write modes, and directory quotas

The following subroutines are described in this chapter. See Chapter 7 of this volume for descriptions of OPSR$ and OPSRS$, which use search rules to locate and open files.

APSFX$	Append a specified suffix to a pathname.
CF$EXT	Extend or truncate a CAM file.
CF$REM	Retrieve a CAM file's extent map from disk.
CF$SME	Set a CAM file's allocation size value.
CH$MOD	Change the open mode of an open file.
CL$FNR	Close a file by name and indicate closed units.

CLO$FN	Close a file system object by pathname.
CLO$FU	Close a file system object by file unit number.
CNAM$$	Change the name of an object in the current directory.
CREA$$	Create a new subdirectory in the current directory.
CREPW$	Create a new password directory.
DIR$CR	Create a new directory.
DIR$LS	Search for specified types of entries in directory open on file unit.
DIR$RD	Read sequentially entries of directory open on file unit.
DIR$SE	Return entries meeting caller–specified selection criteria in a directory open on a file unit.
ENT$RD	Return contents of entry in directory open on file unit.
EQUAL$	Generate a filename based on another name.
EXTR$A	Return file system object's entryname and parent directory pathname.
FIL$DL	Delete a file identified by a pathname.
FINFO$	Return information about a specified file unit.
FNCHK$	Verify a supplied string as a valid filename.
FORCEW	Force PRIMOS to write modified records to disk.
GPATH$	Return pathname of specified unit, attach point or segment.
ISREM$	Determine whether an open file system object is local or remote.
LDISK$	Return information on the system's list of logical disks.
LUDSK$	List the disks a given user is using.
NAM$AD_PORTAL	Converts an existing directory entry into a portal by mounting the defined portal over the directory.
NAM$L_GMT	Reads the contents of the Global Mount Table (GMT) and lists both the currently mounted disk partitions and the currently mounted portals which can be accessed by the calling program.
NAM$RM_PORTAL	Deletes a portal entry in the specified directory pathname.

PAR$RV	Return a logical value indicating whether a specified partition supports ACL protection and quotas.
PRWF$$	Read, write, position, or truncate a file.
Q$READ	Return directory quota and disk record use information.
Q$SET	Set a quota on a subdirectory in the current directory.
RDEN$$	Position in or read from a directory.
RDLIN$	Read a line of characters from an ASCII disk file.
SATR$$	Set or modify an object's attributes in its directory entry.
SGD$DL	Delete a segment directory entry.
SGD$EX	Determine if a segment directory entry exists.
SGD$OP	Open a segment directory entry.
SGDR$$	Position in, read an entry in, or modify the size of a segment directory.
SIZE$	Return the size of a file system entry.
SRCH$$	Open, close, delete, change access, or verify the existence of an object
SRSFX$	Search for a file with a list of possible suffixes.
TNCHK$	Verify a supplied string as a valid pathname.
TSRC$$	Open a file anywhere in the PRIMOS file structure.
UNITS$	Return the minimum and maximum file unit numbers currently in use by this user.
WILD$	Return a logical value indicating whether a wildcard name was matched.
WTLIN$	Write a line of characters to a file in compressed ASCII format.

APSFX$

APSFX$ appends a specified suffix to a pathname.

Usage

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

CALL APSFX$ (in_pathname, out_pathname, suffix, code);
```

Parameters

in_pathname
INPUT. Pathname input to check for suffix (128 characters maximum).

out_pathname
OUTPUT. Pathname returned to caller with desired suffix appended (128 characters maximum).

suffix
INPUT. This is the suffix to be added to the pathname. It should include the period, and be in capital letters, for example, .F77 (32 characters maximum).

code
OUTPUT. Standard error code. Possible values are

–1	Suffix already present, pathname remained unchanged.
E$OK	Suffix appended successfully.
E$NMLG	Pathname added to suffix is more than 128 characters or filename added to suffix is longer than 32 characters.

Discussion

The APSFX$ subroutine is designed for use with the object–naming convention that appends suffixes to an object name by means of a period, such as MYPROG.CBL. (Refer to the *PRIMOS User's Guide* for a discussion of suffixes.) The pathname is checked for the prior existence of the suffix to avoid overwriting an existing object.

APSFX$ does not permanently change the name of the object; it changes only the name returned in *out_pathname*. It is most often used after an SRSFX$ call.

After SRSFX$ finds an object and determines its suffix, APSFX$ can be used to add a suffix to the base name found in order to generate a name for a related file.

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

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

CH$MOD

CH$MOD changes the open mode of an open file.

Usage

DCL CH$MOD ENTRY (FIXED BIN, FIXED BIN, FIXED BIN);

CALL CH$MOD (***key, unit, code***);

Parameters

key

INPUT. Mode to be set. Possible values are

K$READ	Read (input only) mode
K$WRIT	Write (output only) mode
K$RDWR	Read/write (input/output) mode

unit

INPUT. File unit number on which file whose mode is to be changed is open.

code

OUTPUT. Standard error code.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CL$FNR

CL$FNR closes a file by name and return a bit string indicating closed units.

Usage

```
DCL CL$FNR ENTRY (CHAR(128) VAR,
                  1, 2 FIXED BIN (15),
                     2 (*) BIT (16) ALIGNED,
                  FIXED BIN, FIXED BIN);
```

CALL CL$FNR (*pathname, rtn_list, first_file_unit, code*);

Parameters

pathname

INPUT. Pathname of object to be closed.

rtn_list

OUTPUT. Bit string indicating file units closed, relative to *first_file_unit.*

first_file_unit

OUTPUT. Lowest file unit number closed by this call.

code

OUTPUT. Standard error code.

Discussion

The CL$FNR subroutine closes all of the open file units associated with the file name specified in *pathname.* The bit string returned in *rtn_list* indicates the file unit numbers closed relative to the number returned in *first_file_unit.*

For example, if file units 31, 36, and 40 were open and associated with the file named in *pathname*, then *first_file_unit* returns 31, and the *rtn_list* returns the bit string 1000010001. The first 1–bit represents file unit 31, the next 1–bit represents file unit 36, and the final 1–bit file unit 40.

The intervening file unit numbers 32–35 and 37–39 were either not open or not associated with *pathname*, and hence were not closed by this call.

The UNITS$ call, described later in this chapter, can be used to determine the highest open unit number, and hence the size of *rtn_list.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CLO$FN

CLO$FN closes a file system object by pathname.

Usage

DCL CLO$FN ENTRY (CHAR(128) VAR, FIXED BIN);

CALL CLO$FN (*pathname, code*);

Parameters

pathname
INPUT. Pathname of object to be closed.

code
OUTPUT. Standard error code.

Discussion

The CLO$FN call closes one or more file units associated with the object named in *pathname*. Only file units opened by the calling user are closed. Unlike the CL$FNR call described earlier in this chapter, the identities of the file units closed are not returned.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CLO$FU

CLO$FU closes a file system object by file unit number.

Usage

DCL CLO$FU ENTRY (FIXED BIN, FIXED BIN);

CALL CLO$FU (*unit, code*);

Parameters

unit

INPUT. File unit number to close.

code

OUTPUT. Standard error code.

Discussion

The CLO$FU call closes only the file unit specified in *unit*, regardless of how many file units may be associated with the same object. That is, if the file MYFILE is open on file units 31, 36, and 40, and a CLO$FU call is issued for file unit 36, only the instance of MYFILE that is open on file unit 36 is closed.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CNAM$$

CNAM$$ changes the name of an object in the current directory.

Usage

DCL CNAM$$ ENTRY (CHAR(32), FIXED BIN, CHAR(32),
FIXED BIN, FIXED BIN [, FIXED BIN]);

CALL CNAM$$ (*oldnam, oldlen, newnam, newlen, code* [, *ok_open*]);

Parameters

oldnam
INPUT. Name of the file to be changed.

oldlen
INPUT. Length in characters of *oldnam.*

newnam
INPUT. New name of the file.

newlen
INPUT. Length in characters of *newnam.*

code
OUTPUT. Standard error code.

ok_open
OPTIONAL INPUT. Permits name changing on an open file. Set to 1 to enable this function, otherwise omit. Valid only when *oldnam* and *newnam* are of equal length.

Discussion

The user must have Delete and Add access to the parent directory of the object to change the object's name.

CNAM$$ does not change the date/time last modified (DTM) or the date/time last accessed (DTA) or any of the other attributes of the object. However, the DTM and DTA of the directory in which the object resides are changed. CNAM$$ causes the position of the object's name in its parent directory to change with respect to those of other objects if the new name is longer than the old name.

It is invalid to attempt to change the name of the MFD, BOOT, or BADSPT objects. An E$NRIT error message is generated if this is attempted.

Ordinarily, changing the name of an object is done only while the object is closed. However, it is possible, by means of the *ok_open* parameter, to change an object's name while the file is open, provided the old name and the new name are equal in length. If they are not, and the value of *ok_open* is 1, an error is returned.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

CREA$$

CREA$$ creates a new subdirectory in the current directory.

This subroutine is considered obsolete, and its use in new programming is discouraged. Use DIR$CR instead. Users maintaining existing programs that call CREA$$ can refer to Appendix A for a complete description of the subroutine.

CREPW$

CREPW$ creates a new password directory.

This subroutine is considered obsolete, and its use in new programming is discouraged. Use DIR$CR instead. Users maintaining existing programs that call CREPW$ can refer to Appendix A for a complete description of the subroutine.

DIR$CR

DIR$CR$ creates a new directory.

Usage

DCL DIR$CR ENTRY (CHAR(128) VAR, POINTER, FIXED BIN(15));

CALL DIR$CR (*pathname, attribute_pointer, code*);

Parameters

pathname

INPUT. Pathname of the directory to be created.

attribute_pointer

INPUT. Pointer to a program–declared block of attributes to be given to the new directory. The attribute structure is described below.

code

OUTPUT. Standard error code.

Discussion

The DIR$CR call replaces the obsolete subroutines CREA$$ and CREPW$.

DIR$CR allows you to create an ACL directory or a password directory anywhere in the file system. The caller must have Add permission to the parent directory.

If the pathname parameter is an entryname (that is, it contains no > characters), the directory is created at the current attach point.

The structure pointed to by *attribute_pointer* is expected to have the following declaration (all elements are input):

```
DCL 1 attributes,
      2 version FIXED BIN(15),
      2 dir_type FIXED BIN(15),
      2 max_quota FIXED BIN(31),
      2 access_cat CHAR(32)VAR;
```

version

Structure version number. Currently must be 1.

dir_type

Type of directory to create. Possible values are

K$SAME	New directory has same type as the parent directory.
K$PWD	New directory is a password directory. Owner and nonowner passwords are set to their defaults of spaces and nulls, respectively.

max_quota

Maximum quota for new directory. The disk must be a quota disk.

access_cat

Entryname for an access category by which the new directory will be protected (input). Not permitted if the parent directory is a password–protected directory.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

DIR$LS

DIR$LS searches for specified types of entries in a directory open on a file unit.

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*);**

Parameters

dir_unit

INPUT. Unit on which the directory to be searched is open.

dir_type

INPUT. Type of object open on *dir_unit*. Possible values are

2	SAM segment directory
3	DAM segment directory
4	User directory

initialize

INPUT. If set, the directory is to be reset to the beginning; otherwise, it is searched from the current position. This enables large directories to be dealt with in more than one call, making a large buffer area in the calling program unnecessary.

desired_types

INPUT. A bit–encoded field defining what types of directory entries the caller wants to have returned. In the following table, if the bit is set the specified type is 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

INPUT. Pointer to list of wildcard names for which to search. The list is an array of CHAR(32) varying strings; the wildcard names must be uppercase. Wildcards are explained in the *PRIMOS User's Guide.*

wild_count

INPUT. Number of names in list pointed to by *wild_ptr*. If *wild_count* is 0, wildcards are not used as a selection criterion.

return_ptr

INPUT –> OUTPUT. Pointer to caller's return structure. The data structure returned is declared in the program as described below.

max_entries

INPUT. Maximum number of entries that caller's structure can contain.

entry_size

INPUT. Number of halfwords reserved for each directory entry in the caller's structure. *max_entries* multiplied by *entry_size* defines the size of the caller's structure in halfwords. In Rev. 20.2, the normal size of a returned directory entry is 31 halfwords.

ent_returned

OUTPUT. Number of entries returned in the current call. This number is always less than or equal to *max_entries.*

type_counts

OUTPUT. Number of entries of each type returned. Counts are returned in the order of files, segment directories, directories, access categories, the sum of all giving the current total number of entries. At Rev. 20.2, they are reset to zero when the initialize bit is set.

before_date

INPUT. Entries with date/time modified *earlier* than this date are selected. The date is given in standard FS format, described below. If the value of *before_date* is 0, it is not used as a selection criterion.

after_date

INPUT. Entries with date/time modified *later* than this date are selected. The date is given in standard FS format, described below. If the value of *after_date* is 0, it is not used as a selection criterion.

code

OUTPUT. 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 file directory scanner. It selects directory entries by name (handling wildcards), type, and date/time modified (DTM). It can also be used to search segment directories.

The directory must have been previously opened on some unit with one of the standard PRIMOS object–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  date_time_mod FIXED BIN(31),
       2  non_default_acl BIT(1) ALIGNED,
       2  logical_type FIXED BIN,
       2  trunc BIT(1) ALIGNED,
       2  date_time_backed_up FIXED BIN(31);
```

ecw.type

Entry control word for the entry. Possible values are

2	Normal directory entry (file, directory, or segment directory)
3	An access category

ecw.length

24 halfwords for PRIMOS revisions up to and including 19.2, 27 halfwords for revisions from 19.3, 31 halfwords from Rev. 20.0, and 37 halfwords from Rev. 22.0 onward.

entryname

Name of the entry, in uppercase. If a root directory has been designated as a portal, with the NAM$AD_PORTAL subroutine, *entryname* refers to the portal.

protection.owner_rights

The rights granted to a user when attached to the containing directory, having given the owner password.

protection.delete_protect

The setting of the ACL delete–protect switch. If this bit is on, the file cannot be deleted. The bit can be reset by a call to the SATR$$ subroutine.

protection.non_owner_rights

The rights granted to a user when attached to the containing directory, having given the nonowner password or no password.

file_info.long_rat_hdr

If set, indicates that the file is a Disk Record Availability Table (DSKRAT) containing more than one record.

file_info.dumped

If set, the file has been backed up by MAGSAV.

file_info.dos_mod

If set, the file was modified while PRIMOS II (DOS) was running. This applies only to files on pre–Rev. 20.0 disks.

file_info.special

If set, the file is special (for example, DSKRAT, BOOT, MFD) and cannot be deleted. For the root entry, this is always set.

file_info.rwlock

Indicates the setting of the file's read/write concurrency lock, which can be set with the PRIMOS RWLOCK command. Possible values are

0	Use system default setting (SYS option).
1	Unlimited readers or one writer (EXCL option).
2	Unlimited readers and one writer (UPDT option).
3	Unlimited readers and writers (NONE option).

file_info.spare

Two bits presently undefined.

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
7	CAM file

date_time_mod

The date/time the file was last modified, in standard FS format. FS–format dates are described in Appendix C of Volume III. For the root directory (<), this information is returned for the date last *mounted.*

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. For the root directory (<), this attribute requires a reference to the MFD of the mounted disk.

logical_type

This is an additional file type to the physical file type described in *file_info.type*. Possible values are

0	Normal file
1	Recovery based file (RBF)

trunc

This bit is set if the file has been truncated by the FIX_DISK utility; otherwise, it is zero.

date_time_backed_up

Reserved for future use. This field is currently returned as zero (unset).

Valid *dir_entry* Attributes for the Root Directory: The following are the valid attributes for the root directory (<), except when remote ADDISK operations are used to build the root. In that case, all attributes will be set to either a default value or a nonapplicable value.

- *owner_rights*, *delete_protect*, *non_owner_rights*, *file_info*, and *dtb* attributes are read at mount time only.
- The *dtm* attribute is the time of the logical–mount (ADDISK) operation.
- The *non_default_acl* is always set for a nonpassword MFD.
- The *dta* is not set.
- The *access* and *protected_by* attributes require a reference to the MFD of the mounted disk.

The root directory has no actual attributes because the root directory has no entry itself in another directory; it is at the top of the common file system name space. However, using a portal, you can make it appear as if the root were contained in another directory. In this case, only the following attributes are set by the DIR$ subroutines:

ecw.type	Directory
ecw.length	37 halfwords
entryname	The name of the portal
special	Set
non_default_protection	Set (the root directory always has $REST:LU ACL rights)

For more information on the root directory, refer to the discussion on the Rev. 23.0 file system in the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

DIR$RD

DIR$RD reads sequentially the entries of a directory open on a file unit.

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*);

Parameters

key

INPUT. Indicates whether to initialize for subsequent reading or to read from current position. Possible values are

K$INIT	Initialize to directory header.
K$READ	Read from current position.

unit

INPUT. Unit number on which directory is open. User must have List access to the directory.

return_ptr

INPUT -> OUTPUT. Pointer to program–declared directory structure (described below).

max_return_len

INPUT. Size of user's buffer.

code

OUTPUT. Standard error code.

Discussion

FS–format dates are structured as described in Appendix C of Volume III.

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.

The *return_ptr* points to a directory entry structure with the following format. Note that *ecw* is actually a halfword.

```
DCL 1  dir_entry BASED,
       2  ecw,
          3  type BIT(8),
          3  length BIT(8),
       2  entryname CHAR(32),
       2  pw_protection BIT(16) ALIGNED,
       2  non_default_protection BIT(1) ALIGNED,
       2  file_info,
          3  long_rat_hdr BIT(1),
          3  dumped_bit BIT(1),
          3  dos_mod BIT(1),
          3  special BIT(1),
          3  rwlock BIT(2),
          3  reserved BIT(2),
          3  type BIT(8),
       2  date_time_modified,
          3  date,
             4  year BIT(7),
             4  month BIT(4),
             4  day BIT(5),
          3  time FIXED BIN,
       2  spare (2) FIXED BIN,
       2  trunc BIT(1) ALIGNED,
       2  dtb like date_time_modified,
       2  dtc like date_time_modified,
       2  dta like date_time_modified;
```

ecw.type

Entry control word for the entry. Values are

2	Normal directory entry (file, directory, or segment directory)
3	Access category

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

DIR$RD returns entries only for named objects. Thus it does not return the *ecw* (entry control word) for the directory header. *type* is 2 for a file or directory, and 3 for an access category.

ecw.length

24 halfwords for PRIMOS revisions up to and including 19.2, 27 halfwords for revisions from 19.3, 31 halfwords from 20.0 onward, and 37 halfwords from 22.0 onward.

entryname

The name of the entry, in uppercase, left–justified, and filled with spaces. If a root directory has been designated as a portal, with the NAM$AD_PORTAL subroutine, *entryname* refers to the portal.

pw_protection

Owner and nonowner protection attributes. For the root entry, this attribute is read at mount time only. 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

non_default_protection

Set to true ('1'b) if the entry is *not* default–protected; it is either protected specifically or by an access category. For the root entry, this attribute requires a reference to the MFD of the mounted disk.

file_info.long_rat_hdr

If set, indicates that the file is a Disk Record Availability (DSKRAT) file spanning more than one disk record.

file_info.dumped_bit

If set (=1), this file has been saved by MAGSAV and has not been modified since then.

file_info.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. This applies only to files on pre–Rev. 20.0 disks.

file_info.special

If set, this is a special file (for example, DSKRAT, BOOT, MFD) and cannot be deleted. For the root entry, this is always set.

file_info.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)

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	User directory
6	Access category
7	CAM file

date_time_modified

The date and time, in standard FS format, that the entry was last modified. For the root directory (<), this information is returned for the date last *mounted.*

trunc

This bit is set if the entry has been truncated by the FIX_DISK utility; otherwise, reset to zero.

dtb

Date and time the file was last backed up. For the root directory (<), *dtb* is read at mount time only.

dtc

Date and time the file was last created.

dta

Date and time the file was last accessed. This attribute is not set for the root directory (<).

Valid *dir_entry* Attributes for the Root Directory: The following are the valid attributes for the root directory (<), except when remote ADDISK operations are used to build the root. In that case, all attributes will be set to either a default value or a nonapplicable value.

- *owner_rights*, *delete_protect*, *non_owner_rights*, *file_info*, and *dtb* attributes are read at mount time only.
- The *dtm* attribute is the time of the logical–mount (ADDISK) operation.
- The *non_default_acl* is always set for a nonpassword MFD.
- The *dta* is not set.
- The *access* and *protected_by* attributes require a reference to the MFD of the mounted disk.

The root directory has no actual attributes because the root directory has no entry itself in another directory; it is at the top of the common file system name space. However, using a portal, you can make it appear as if the root were contained in another directory. In this case, only the following attributes are set by the DIR$ subroutines:

ecw.type	Directory
ecw.length	37 halfwords
entryname	The name of the portal
special	Set
non_default_protection	Set (the root directory always has $REST:LU ACL rights)

For more information on the root directory, refer to the discussion on the Rev. 23.0 file system in the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

DIR$SE

DIR$SE returns entries meeting caller–specified selection criteria to a directory open on a file unit.

Usage

```
DCL DIR$SE ENTRY (FIXED BIN(15), FIXED BIN(15), BIT(1),
                  PTR OPTIONS(SHORT), PTR OPTIONS
                  (SHORT),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***);

Parameters

dir_unit

INPUT. Unit on which directory to be searched is open.

dir_type

INPUT. Type of object open on *dir_unit*. Possible values are

2	SAM segment directory
3	DAM segment directory
4	User directory

initialize

INPUT. If set (='1'b), directory is to be reset to the beginning. If not set, the directory is to be searched from the current position.

sel_ptr

INPUT. Pointer to the structure containing selection criteria. See the Discussion section. In languages that default to using a long pointer, *sel_ptr* must be declared as OPTIONS (SHORT).

return_ptr

INPUT –> OUTPUT. Pointer to caller's return structure for selected entry data. This parameter points to where the subroutine places the output. In languages that default to a long pointer, *return_ptr* must be declared as a short pointer.

max_entries

INPUT. Maximum number of entries to be returned. If the value of *initialize* is 1, and if the call is being used only to initialize the directory search, and not to return any entries, this parameter is 0. The value of *max_entries* should be set to a maximum of 150. If you anticipate more entries than 150, establish a loop that reiterates the call to DIR$SE until the end of file code (E$EOF) is returned. Failure to use this maximum value can cause unpredictable results.

entry_size

INPUT. Number of halfwords to be returned per entry. Permissible values of *entry_size* are given in the *length* entry in the description of the entry control word (see the Discussion section).

ent_returned

OUTPUT. Number of entries returned.

type_counts

INPUT/OUTPUT. Number of entries of each type returned in this order: files, segment directories, directories, access categories. This parameter is a four–halfword array. 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.

max_type

INPUT. Number of types in *type_counts* (currently must be 4).

code

OUTPUT. Standard error code. Possible values are

E$BVER	Invalid version number for selection criteria structure.
E$BPAR	Bad max_type (currently must be four).
E$EOF	There are no more entries in the directory to be selected.
E$ST19	Selection criteria involving recovery based file (RBF) type or date/time last backed up, accessed, or created have been specified, and the PRIMOS revision that accesses the directory does not support these features.
E$NTUD	Object open on *dir_unit* is not a directory.
E$NBUF	No buffer space. Number specified for *max_entries* is too large.

Discussion

The selection criteria should be supplied in one of the following structures. The first field in the structure, *version_no*, indicates which of the two structures the caller is providing. Version 0 is provided for compatibility with Revision 19.4 of the operating system, but can be used if the date/time accessed or date/time created fields are not used as selection criteria. Version 1 should be used for Revision 20.0 and subsequent revisions. The *sel_ptr* parameter should point to the structure.

Version 0

```
DCL 1  selection_criteria BASED,
       2  version_no FIXED BIN,     /* Must be 0 */
       2  wild_ptr PTR OPTIONS(SHORT),
       2  wild_count FIXED BIN,
       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 FIXED BIN(31),
       2  modified_after_date FIXED BIN(31),
       2  backed_up_before_date FIXED BIN(31),
       2  backed_up_after_date FIXED BIN(31);
```

Version 1

```
DCL 1  selection_criteria BASED,
       2  version_no FIXED BIN,     /* Must be 1 */
       2  wild_ptr PTR OPTIONS(SHORT),
       2  wild_count FIXED BIN,
       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 FIXED BIN(31),
       2  modified_after_date FIXED BIN(31),
       2  backed_up_before_date FIXED BIN(31),
       2  backed_up_after_date FIXED BIN(31),
       2  created_before_date FIXED BIN(31),
       2  created_after_date FIXED BIN(31),
       2  accessed_before_date FIXED BIN(31),
       2  accessed_after_date FIXED BIN(31);
```

version_no

Must be 0 for the first version (Version 0) of the selection criteria structure, or 1 for the second version (Version 1).

wild_ptr

If wildcard entryname selection is to be applied to the directory entries, this field points to a list of wildcard names for which to search. The list is an array of CHAR(32) varying strings, and the names must be in uppercase. Wildcards are explained in the *PRIMOS User's Guide* and the *PRIMOS Commands Reference 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 are both set, and the other bits are not set. If the first four bits are set, all entries are returned. If all five bits are set, all entries that are also RBF entries are returned.

The fields listed below select entries based on one of the four date attributes. The input date is in standard FS format, or is zero if this field is not to be used as a selection criterion.

modified_before_date

Selects entries with date/time modified earlier than this date.

modified_after_date

Selects entries with date/time modified later than this date.

backed_up_before_date

Selects entries with date/time backed up earlier than this date. The date/time backed up field is set by the BRMS backup utility.

backed_up_after_date

Selects entries with date/time backed up later than this date.

created_before_date

Selects entries with date/time created earlier than this date.

created_after_date

Selects entries with date/time created later than this date.

accessed_before_date

Selects entries with date/time accessed earlier than this date.

accessed_after_date

Selects entries with date/time accessed later than this date.

FS–format dates are structured as shown in Appendix C of Volume III.

Example

DIR$SE returns 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(2),
          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 BIN,
       2  trunc BIT(1) ALIGNED,
       2  date_time_backed_up FIXED BIN(31),
       2  date_time_created FIXED BIN(31),
       2  date_time_accessed FIXED BIN(31);
```

ecw.type

Entry control halfword for the entry. Values are

2	Normal directory entry (file, file directory, or segment directory)
3	Access category

ecw.length

24 halfwords for PRIMOS revisions up to and including 19.2, 27 halfwords for revisions from 19.3, 31 halfwords from Rev. 20.0 onward, and 37 halfwords from Rev. 22.0 onward.

entryname

Name of the entry, in uppercase. If a root directory has been designated as a portal, with the NAM$AD_PORTAL subroutine, *entryname* refers to the portal.

protection.owner_rights

Rights granted to a user, when attached to the containing directory having owner rights. For the root directory (<), this attribute is read at mount time only.

protection.delete_protect

If this bit is set, the file cannot be deleted. The bit can be reset by a call to the SATR$$ routine. The root directory (<) cannot be deleted.

protection.non_owner_rights

Rights granted to a user, when attached to the containing directory having nonowner rights. For the root directory (<), this attribute is read at mount time only.

file_info.long_rat_hdr

If set, indicates that the file is a Disk Record Availability (DSKRAT) file spanning more than one disk record.

file_info.dumped

If set, this file has been saved by MAGSAV and has not been modified since then.

file_info.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. This applies only to files on pre–Rev. 20.0 disks.

file_info.special

If set, this is a special file (for example, DSKRAT, BOOT, MFD) and cannot be deleted. For the root directory (<), this is always set.

file_info.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)

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	User directory
6	Access category
7	CAM file

date_time_mod

The date/time the file was last modified, in standard file system format. FS–format dates are coded as shown in Appendix C of Volume III. For the root directory (<), *date_time_mod* refers to the date last *mounted.*

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. For the root directory (<), this attribute requires a reference to the MFD of the mounted disk.

logical_type

This is an additional file type to the physical file type described in *file_info.type*. Possible values are

0	Normal files
1	RBF files

trunc

This bit is set if the file has been truncated by the FIX_DISK utility; otherwise, reset to zero.

date_time_backed_up

This field returns the date and time the file was last saved by the BRMS backup utility, in FS format. If it has never been saved, the value is zero. For the root directory (<), *date_time_backed_up* is read at mount time only.

date_time_created

On a Rev. 20.0 partition, this field returns the date and time the file was created in FS format. On a Revision 19.0 partition, the returned value is zero.

date_time_accessed

On a Rev. 20.0 partition, this field returns the date and time the file was last accessed, in FS format. On a Revision 19.0 partition, the returned value is zero. This attribute is not set for the root directory (<).

Valid *dir_entry* Attributes for the Root Directory: The following are the valid attributes for the root directory (<), except when remote ADDISK operations are used to build the root. In that case, all attributes will be set to either a default value or a nonapplicable value.

- *owner_rights*, *delete_protect*, *non_owner_rights*, *file_info*, and *dtb* attributes are read at mount time only.
- The *dtm* attribute is the time of the logical–mount (ADDISK) operation.
- The *non_default_acl* is always set for a nonpassword MFD.
- The *dta* is not set.
- The *access* and *protected_by* attributes require a reference to the MFD of the mounted disk.

The root directory has no actual attributes because the root directory has no entry itself in another directory; it is at the top of the common file system name space. However, using a portal, you can make it appear as if the root were contained in another directory. In this case, only the following attributes are set by the DIR$ subroutines:

ecw.type	Directory
ecw.length	37 halfwords
entryname	The name of the portal

special	Set
non_default_protection	Set (the root directory always has $REST:LU ACL rights)

For more information on the root directory, refer to the discussion on the Rev. 23.0 file system in the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

ENT$RD

ENT$RD returns the contents of a named entry in a directory open on a file unit.

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*);

Parameters

unit

INPUT. Unit number on which the directory is open.

name

INPUT. Name of the entry to read.

return_ptr

INPUT –> OUTPUT. Pointer to program–declared return structure.

max_return_len

INPUT. Size of user's buffer.

code

OUTPUT. 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$RD is identical to that described for the DIR$RD subroutine.

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.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EQUAL$

EQUAL$ generates a filename based on another name.

Usage

```
DCL EQUAL$ ENTRY (CHAR(32) VAR, CHAR(32) VAR,
                  CHAR(32) VAR, FIXED BIN(15));

CALL EQUAL$ (obj_name, pattern, generated, code);
```

Parameters

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. Standard error code.

Discussion

This routine expects an objectname and a generation pattern. The latter contains "commands" that specify how to transform the objectname into a new name called the generated name. This routine performs that transformation. Name generation is discussed in the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EXTR$A

EXTR$A returns a file system object's entryname and parent directory pathname.

Usage

```
DCL EXTR$A ENTRY (CHAR (*) VAR, CHAR (*) VAR,
                  FIXED BIN (15), CHAR (32) VAR, FIXED BIN (15));

CALL EXTR$A (full_path, parent_path, max_length, entryname, code);
```

Parameters

full_path

INPUT. Object's pathname that is to be split into a parent directory pathname and an entryname.

parent_path

OUTPUT. Object's parent directory pathname.

max_length

INPUT. Maximum length of *parent_path* in characters.

entryname

OUTPUT. Last element of *full_path* (the part of *full_path* that follows the last > symbol).

code

OUTPUT. Standard error code. Possible values are

E$BPAR	*full_path* is not a legal pathname.
E$BFTS	The returned length of the parent directory pathname is greater than *max_length*.

Discussion

Given the full pathname of a file system object, the EXTR$A subroutine separates the pathname of the directory that immediately contains the object from the entryname of the object, and returns them as two separate elements. Your program can then do any appropriate directory operations on the name returned in *parent_path*, and any appropriate file operations on the name returned in *entryname*.

Note At Rev. 23.0, EXTR$A acts as follows when referencing the root directory.

input	*parent_path*	*entryname*
<	null	null
<dir_name	<	dir_name
<dir_name>mfd	<dir_name>mfd	mfd

You cannot use EXTR$A to access the root directory via the MFD. Therefore, the MFD must be treated as a special case in your program if you want to access the root.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

FIL$DL

FIL$DL deletes a file identified by a pathname.

Usage

DCL FIL$DL ENTRY (CHAR(128)VAR, FIXED BIN);

CALL FIL$DL (*object_name, code*);

Parameters

object_name

INPUT. Pathname of the object to be deleted.

code

OUTPUT. Standard error code. 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 can be deleted. This error code is returned only if the caller has Delete access on the parent directory and is thus allowed to reset the delete–protect switch.

Deleting an object returns its records to the DSKRAT pool of free records and erases the entry from the directory, leaving a hole. Holes in directories are reused for new objects if they are large enough to contain the new object's name, so new objects do not always appear at the end of a directory. Holes take very little room on the disk in most cases. They are compressed out of directories when the

FIX_DISK maintenance program is run by the system operator. FIX_DISK is described in the *Operator's Guide to File System Maintenance.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

FINFO$

FINFO$ returns information about a specified file unit.

Usage

DCL FINFO$ ENTRY (FIXED BIN, PTR, FIXED BIN);

CALL FINFO$ (***unit, finfo_ptr, code***);

Parameters

unit

INPUT. File unit on which object whose information is wanted is open.

finfo_ptr

INPUT –> OUTPUT. Pointer to user–declared structure in which information is to be returned.

code

OUTPUT. Standard error code.

Discussion

The FINFO$ call returns information about the object open on the specified file unit (or attach point), including

- Open mode (Read, Read/Write, VMFA read, Attach point)
- Status info (remote, modified, etc.)
- Position
- Read/write lock
- File type
- Logical device number

File information is returned in a structure pointed to by *finfo_ptr* and formatted as shown below.

```
DCL 1 finfo_ BASED,
       2 version FIXED BIN(15),    /* Must be 1 or 2. */
       2 status,
          3 modified BIT (1),
          3 remote BIT (1),
          3 shut_down BIT (1),
          3 no_close BIT (1),
          3 disk_error BIT (1),
          3 spare1 BIT (3),
       2 open_mode,
          3 spare2 BIT (3),
          3 vmfa_read BIT (1),
          3 block_mode BIT (1),
          3 attach_point BIT (1),
          3 write BIT (1),
          3 read BIT (1),
       2 file_type FIXED BIN(15),
       2 rwlock FIXED BIN(15),
       2 position FIXED BIN (31),
       2 system_name CHAR(32) VAR,
       2 ldevno FIXED BIN(15),
       2 packname CHAR(32) VAR,
       2 BRA fixed bin(31),
       2 ldev fixed bin;
```

version

INPUT. The version number of the data structure. The user program must specify 1 or 2 to select a version. Version 2 is identical to Version 1 except that it includes two additional fields, *BRA* and *ldev* (see below).

status

OUTPUT. Information about the current status of the file. The bits have the following meaning when set:

modified	The file has been modified.
remote	Thc file is remote.
shut_down	The disk has been shut down.
no_close	The file may not be closed.
disk_error	There is a disk error on this file.
spare1	Reserved.

open_mode

OUTPUT. The bits have the following meaning when set:

spare2	Reserved.
vmfa_read	Open for VMFA read (subkey K$VMR).
block_mode	Open for block mode (subkey K$BKIO).
attach_point	Current or home attach point.
write	Open for write.
read	Open for read.

For information about the use of the subkeys K$VMR and K$BKIO, see the description of SRCH$$ later in this chapter.

file_type

OUTPUT. Indicates type of file. The possible values are

0	SAM
1	DAM
2	SAM segment directory
3	DAM segment directory
4	User directory
5	ACL directory
7	CAM

rwlock

OUTPUT. Read/write lock. The possible values are

0	Open to 1 reader or 1 writer (subkey K$DFLT).
1	Open to any number of readers *or* 1 writer (subkey K$EXCL).
2	Open to any number of readers *and* 1 writer (subkey K$UPDT).
3	Open to any number of readers or writers (subkey K$NONE).

For information about the use of subkeys to set file attributes, see the description of SATR$$ later in this chapter.

position

OUTPUT. Read/write pointer position.

system_name

OUTPUT. System name for remote unit. If the unit is not remote, returns a null string.

ldevno

OUTPUT. Logical device of unit from perspective of local node.

packname

OUTPUT. Packname of partition.

BRA

OUTPUT. File's beginning record address. Returned only when the Version 2 structure is selected. (See above.)

ldev

OUTPUT. Logical device for unit for remotely managing node if a remote object. Returned only when the Version 2 structure is selected. (See above.)

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

FNCHK$

FNCHK$ verifies a supplied string as a valid filename.

Usage

```
DCL FNCHK$ ENTRY (FIXED BIN, CHAR(*)VAR)
                         RETURNS (BIT(1));
```

name_ok = FNCHK$ (*key, filename*);

Parameters

key

INPUT. Defines restrictions on *filename*. Keys can be added together; for example, K$UPRC+K$WLDC. Possible values are

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 entrynames).

filename

INPUT/OUTPUT. Name to be checked (input only unless K$UPRC is used; in that case, input/output).

name_ok

RETURNED VALUE. Set to true (1) if the name is valid given the restrictions of the keys.

Discussion

This function call validates the string passed as a filename. This means that the string must not contain PRIMOS reserved characters, lowercase letters, or control characters, must not start with a digit, and must be between 1 and 32 characters long. The *key* passed to FNCHK$ can modify these restrictions.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

FORCEW

FORCEW forces PRIMOS to write modified records to disk.

Usage

DCL FORCEW ENTRY (FIXED BIN, FIXED BIN[, FIXED BIN]);

CALL FORCEW (*ignored, funit* [,*code*]);

Parameters

ignored

This parameter is not used. Must be 0.

funit

INPUT. The file unit on which a file has been opened.

code

OPTIONAL OUTPUT. Standard error code. The following codes are added at Rev. 22.0:

E$DISK	A disk error occurred on the file referenced by *funit.* If *code* is not supplied as an argument, then disk errors are not reported.
E$ZERO	Indicates that the system found and zeroed out an uninitialized block in a file on a robust partition.

Discussion

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, because 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.

FORCEW also writes the following items to disk:

- Index blocks of DAM files
- Extent maps of CAM files
- Modified DSKRAT blocks of files on standard partitions, if the files have been extended physically

Note that on robust partitions, it is not necessary to write DSKRAT blocks to disk. For this reason, thc performance of FORCEW may be better on robust partitions.

FORCEW returns information about 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 the *code* argument is not supplied, no action is taken and the error mark on the open file is not reset. The error mark can then be tested 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.

The E$ZERO error can result from improper shutdown of a disk. When a system halt occurs while a file is being extended logically on a robust partition, some data blocks may not be written to disk. As a result, there may be uninitialized blocks in the file. FIX_DISK –FAST will not detect the uninitialized blocks because it does not check the validity of the data in the file. When the user tries to read the uninitialized blocks at runtime, PRIMOS determines that the blocks are invalid, zeros them, and returns E$ZERO. E$ZERO isreturned only the first time that the block is read or part of the block is written. FORCEW can be used to ensure that all data blocks are written to a file so that uninitialized blocks will not occur on the partition.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

GPATH$

Return the pathname of a specified unit, attach point, or segment.

Usage

```
DCL GPATH$ ENTRY (FIXED BIN, FIXED BIN, CHAR (*),
                  FIXED BIN, FIXED BIN, FIXED BIN);

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

Parameters

key

INPUT. Specifies the pathname to be returned. Possible values are

K$UNIT	Pathname of file open on unit specified by *funit* is to be returned.
K$CURA	Pathname of current attach point is to be returned.
K$HOMA	Pathname of home attach point is to be returned.
K$INIA	Pathname of initial attach point (origin) is to be returned.
K$COMO	Pathname of Command Output file is to be returned.
K$SEGN	Pathname of EPF mapped to *funit* is to be returned.

funit

INPUT. Specifies file unit number if *key* is K$UNIT, segment number if *key* is K$SEGN; otherwise ignored.

buffer

OUTPUT. The declared name of the character string in which the pathname is to be returned.

bufflen

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

pathlen

OUTPUT. Specifies the length in characters of the pathname returned in *buffer*.

code

OUTPUT. Standard error code. Possible values are

E$BKEY — A bad *key* was specified, or segment number was out of range.

E$BUNT — A bad unit number was specified in *funit.*

E$UNOP — Unit specified in *funit* is closed; no filename is associated with the unit.

E$NATT — Not attached to any directory (*keys* K$CURA, K$HOMA).

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

E$FNTF — No EPF is mapped to segment *funit.*

Discussion

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.

Note In certain cases at Rev. 23.0 and later revisions, it is possible that GPATH$ may not return the desired pathname. For example, if GPATH$ encounters a remote portal reference to another directory, and that reference has not been propagated to the Global Mount Table, the error `PATH UNAVAILABLE` is returned.

If *key* is K$SEGN, *funit* is interpreted as a segment number. In this case GPATH$ returns the name of the EPF mapped to the segment, if there is one.

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

```
<disk_name>MFD
<SPOOLD>MFD

<disk_name>dir_name
<SPOOLD>SPOOLQ

<disk_name>dir_name1>dir_name2>file_name
<SALESD>WEST.COAST>YTD.1990>MARCH

<disk_name>dir_name>segment_directory_name
<OPSYST>PR4.64>VPRMOS

<disk_name>dir_name>segment_directory_name>entry_num>entry_num
<DBDISK>DICTIONARY>WORDS>22>68
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

ISREM$

Determines whether an open file system object is local or remote.

Usage

```
DCL ISREM$ ENTRY (FIXED BIN, CHAR (128) VAR, FIXED BIN,
          CHAR (32) VAR, FIXED BIN) RETURNS (BIT (1));

is_remote = ISREM$ (key, pathname, unit, sysname, code);
```

Parameters

key

INPUT. Specify how to search for object. Possible values are

K$NAME	Search for object by pathname.
K$UNIT	Search for object by file unit number.

pathname

INPUT. Pathname of object to search for, if *key* is K$NAME.

unit

INPUT. Unit on which object is open, if *key* is K$UNIT.

sysname

OUTPUT. Name of system on which object was found, if remotely attached. Null if object is found on local system.

code

OUTPUT. Standard error code.

is_remote

RETURNED VALUE. Set to TRUE ('1'b) if the object is remotely attached, FALSE ('0'b) if locally attached.

Discussion

The ISREM$ subroutine determines the location (local or remote) of a file system object identified by either its pathname or its file unit number. An error is returned if the object was not previously opened.

If the object is associated with a remote system, ISREM$ returns a bit(1) aligned value of '1'b in *is_remote*; otherwise it returns a value of '0'b. If the object is found to be remote, the system name of the remote node on which the object exists is also returned.

If K$NAME is specified and the path is not the current attach point, the current attach point is set to the home directory when the call is complete.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

LDISK$

Returns information on the system's disk table.

Usage

**DCL LDISK$ ENTRY (FIXED BIN, CHAR(32) VAR, PTR, FIXED BIN,
FIXED BIN);**

CALL LDISK$ (*key, system_name, return_ptr, max_entries, code*);

Parameters

key

INPUT. Indicates what subset of the disk list is desired. Possible values are

K$ALL	All disks
K$LOCL	Local disks only
K$REM	Remote disks only
K$SYS	Disks for specified system only

system_name

INPUT. Name of the system whose disks are desired. Ignored unless *key* is K$SYS.

return_ptr

INPUT -> OUTPUT. Pointer to return structure (defined below).

max_entries

INPUT. Indicates the maximum number of disk information entries that the caller's structure can contain. At Rev. 21.0 and later versions, no more than 238 disk information entries are returned by this routine.

code

OUTPUT. Standard error code. Possible values are

E$BKEY	An illegal key value was passed.
E$BVER	Invalid version number for *disk_list.*
E$BPAR	*max_entries* was less than zero.
E$ROOM	More than *max_entries* disks are in the disk table. This is a warning; data for *max_entries* disks is returned.

Discussion

Depending on the key specified, the LDISK$ subroutine returns information on all disks, local disks only, remote disks only, or disks from a specified remote system.

Information returned includes name, logical device number (local disks only), physical device number, system name if remote, priority ACL status, and write–protect status.

Note At Rev. 23.0, this subroutine is of limited usefulness because LDISK$ returns only manually added disks. Therefore, if Name_Server is running you should use NAM$L_GMT instead to get a complete let of accessible disks.

Declare the structure to which LDISK$ returns information as follows:

```
DCL 1 disk_list BASED,
      2 version FIXED BIN,  /* Must be 1 */
      2 count FIXED BIN,
      2 info (max_ldevs),
        3 p_acl BIT (1),
        3 protected BIT (1),
        3 robust BIT(1),
        3 rsvd BIT (13),
        3 ldevno FIXED BIN,
        3 pdevno FIXED BIN,
        3 disk_name CHAR (32) VAR,
        3 system_name CHAR (32) VAR;
```

version

INPUT. Version number of the structure. The calling program must supply this value. Currently, must be 1.

count

OUTPUT. Number of entries returned in the *info* array (described next). *count* is always equal to the smallest of the following three quantities: the number specified in *max_entries*, the number of disks on the system, or 238.

info.p_acl

OUTPUT. Set if a priority ACL is in effect on this partition. Valid only for local partitions.

info.protected

OUTPUT. Set if the disk is write–protected. Valid only for local partitions.

info.robust

OUTPUT. Set (= 1) if the partition is robust. Not set (= 0) if the partition is not robust.

info.ldevno

OUTPUT. Logical device number of the partition. At Rev. 23.0, this information is returned for the disks on your machine only.

info.pdevno

OUTPUT. Physical device number of the partition.

info.disk_name

OUTPUT. Name of the partition. Currently, a partition name is never more than 6 characters long, but space for 32 is reserved.

info.system_name

OUTPUT. Name of the system on which the disk is physically added. Null for local disks. Currently, this name is 1 – 6 characters long, but space for 32 is reserved. At Rev. 23.0, if *system_name* is a remote system, *info.system_name* is obtained from the Global Mount Table (GMT).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

LUDSK$

Lists the disks a given user is using.

Usage

DCL LUDSK$ ENTRY (FIXED BIN, PTR, FIXED BIN, FIXED BIN);

CALL LUDSK$ (*user, return_ptr, max_entries, code*);

Parameters

user

INPUT. User number whose disks are to be listed. Use 0 (zero) to list disks in use by current user.

return_ptr

INPUT -> OUTPUT. Pointer to structure containing the returned disk list (described below).

max_entries

INPUT. Maximum number of disk entries that the structure can contain. At Rev. 20.2 and later revisions, the maximum is 62.

code

OUTPUT. Standard error code.

Discussion

The LUDSK$ subroutine returns the partition name, the logical device number, and, if a disk is remotely attached, the system name of each disk that is currently in use by the user whose user number is specified in *user*. If *user* is specified as zero, information for the disks in use by the calling user is returned.

Note At Rev. 23.0, only the LDEV pathname syntax (<0>CMDNC0) uses the disk table; all other disk operations go through the Global Mount Table (GMT). LUDSK$ returns a list of disk names, LDEVs, and remote system units that you are using. Since remote disk partitions no longer use LDEVs, the LDEV field that LUDSK$ returns is valid only for local disks.

The structure pointed to by *return_ptr* has the following format:

```
DCL 1 rtn_struc BASED,
       2 version FIXED BIN,
       2 count FIXED BIN,
       2 info (max_devs),
          3 pack_name CHAR(32) VAR,
          3 ldev FIXED BIN,
          3 system_name CHAR(32) VAR;
```

version

Caller–supplied version number of the structure. Must be 2.

count

Number of entries returned. It is the smallest of either the number specified in *max_entries*, number of disks on system, or 62.

info.pack_name

Name of the partition represented by this entry. Currently, all partition names are 1 – 6 characters long, but space for 32 characters is reserved.

info.ldev

Logical disk number associated with this partition. At Rev. 23.0, this information is returned for local disks only.

info.system_name

If the disk in this entry is remote, the system name to which it is physically attached. Currently, system names are 1 – 6 characters long, but space for 32 characters is reserved. If *system_name* is a remote system, this information is obtained from the Global Mount Table (GMT).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

NAM$AD_PORTAL

Converts an existing directory into a portal by mounting the defined portal over the directory. Future references to this directory are redirected to the portal until you remove the portal with NAM$RM_PORTAL, discussed later in this chapter. You may use this subroutine only at the supervisor terminal.

Usage

```
DCL NAM$AD_PORTAL (CHAR(32)VAR, PTR OPTIONS(SHORT),
                         FIXED BIN);
```

NAM$AD_PORTAL (*entryname, portal_info, code*);

Parameters

entryname

INPUT. The entry that is changed when you create the portal.

portal_info

INPUT. The input structure that defines the attributes of the portal you are creating.

code

OUTPUT. The standard return code. Possible values are

E$SCCM	This routine may only be used at the supervisor terminal.
E$BVER	The portal structure version number that you specified is invalid.
E$BKEY	The key that you specified is invalid.
E$NRIT	You do not have access rights for this operation.
E$BNAM	The entryname that you specified uses incorrect syntax.
E$UNOD	The nodename that you specified is not in PRIMENET.
E$BPOR	The portal target must be a remote node.
E$IROO	A portal may not be mounted on a root directory.
E$NTUD	The specified entryname must be a directory.
E$MTPT	A portal already exists at the point where you are trying to mount a portal.
E$RPMH	You cannot create a portal through another portal.

E$IREM	The portal mount must be on a local directory.
E$FNTF	No such entryname exists.

Discussion

The *portal_info* points to an input structure with the following format.

```
DCL 1 portal_info,
     2 version FIXED BIN(15),
     2 portal_target_key FIXED BIN(15),
     2 portal_target,
      3 node_name CHAR(16) VAR, /* must be specified */
      3 partition_name CHAR(6); /* only fordisk_portal*/
```

portal_target_key

INPUT. Type of portal. Possible values are

Key	*Code*	*Meaning*
NAM$K_ROOT	1	root–portal
NAM$K_NOROOT	2	disk–portal

At Rev. 23.0, the new file system introduces the concept of the **portal**, a new file system object that allows you to reference an object on a remote system. A portal acts as a gateway between common file system name spaces, thereby allowing you to perform operations upon file system objects in other name-spaces, transparently.

For a detailed description of the Rev. 23.0 file system, refer to the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action required.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

NAM$L_GMT

Reads the contents of the Global Mount Table (GMT) and returns a list of both the currently mounted disk partitions and the currently mounted portals which the calling program can access.

Usage

DCL NAM$L_GMT (FIXED BIN, POINTER OPTIONS (SHORT),
FIXED BIN(15), FIXED BIN(15), FIXED BIN(15));

CALL NAM$L_GMT (*index, ret_ptr, max_items, ret_items, status*);

Parameters

index

INPUT. A number that indicates the starting Global Mount Table entry in the list to be returned; use *index* when filling in the structure to which *ret_ptr* points. The GMT list of entries may be referenced as an array

[1 ... (*n*–1)]

where *n* is the total of the number of entries in the GMT. Use an array to call the NAM$L_GMT subroutine as many times as there are GMT entries if the declaration of the structure is too small.

ret_ptr

INPUT. A pointer to the structure that NAM$L_GMT fills in (the items for each GMT entry).

max_items

INPUT. The maximum number of entries to be declared as GMT entries in the *index* field. If the *max_items* field is smaller than (*n*–1), structure overflow occurs.

ret_items

INPUT. The number of entries filled in the structure.

status

OUTPUT. The standard return code. (NoError indicates successful completion.)

BadIndex	No such entry exists at the index given in the GMT.

Discussion

Rev. 23.0 introduces the concept of the common file system name space. The contents of the disk partitions within the common file system name space make up *one logical entity*. Therefore, logically, there are no remote disks within the name space. Also, the Global Mount Table for any name space may include as many as 1280 disks, all appearing to be local.

You must be either the System Administrator or User1 (supervisor terminal) to return the remote private partitions (partitions on other machines that were created with the ADDISK –PRIVATE command).

For a detailed description of the Rev. 23.0 file system, refer to the *Advanced Programmer's Guide II: File System.* Refer also to the discussions on the ADDISK command in the *Operator's Guide to System Commands* and in the *System Administrator's Guide, Volume III: System Access and Security.*

The entries in the Global Mount Table are returned by *ret_ptr* to the *gmt_ent_def* data structure, which has the following format:

```
DCL 1 gmt_ent_def BASED,
      2 version FIXED BIN,
      2 owning_node CHAR(16) VAR,
      2 portal_target_node CHAR(16) VAR,
      2 partition_name CHAR(6),
      2 partition_uid CHAR(12),
      2 ldev FIXED BIN,
      2 dtc FIXED BIN(31),
      2 dtm FIXED BIN(31),
      2 state,
         3 portal_entry BIT,
         3 private_partition BIT,
         3 entry_in_root BIT,
         3 pre-ns_entry BIT,
         3 disk_replaced BIT,
         3 remote_disk BIT,
      2 pathname CHAR(128) VAR;
```

version

The version number for the structure.

owning_node

The system to which the object was added.

portal_target_node

The system being referenced by the portal.

partition_name

The name of the disk partition.

partition_uid

The partition UID for the entry.

ldev

The ldev of the disk partition.

dtc

The date and time when the MFD was created.

dtm

The date and time when the MFD was last modified.

state.portal_entry

Set if the entry defines a portal.

state.private_partition

Set if the entry is a private partition.

state.entry_in_root

Set if the partition is added at the root directory.

state.pre_ns_entry

Set if the entry added is from a pre–Rev. 23.0 system.

state.disk_replaced

Set if the disk was added with the –REPLACE option.

state.remote_disk

Set if the disk was added with the –ON option.

pathname

The mount point for the entry.

Loading and Linking Information

V–mode and I–mode: No special action required.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

NAM$RM_PORTAL

Deletes a portal entry in the specified directory pathname. This subroutine may only be used at the supervisor terminal.

Usage

DCL NAM$RM_PORTAL ENTRY (CHAR(32)VAR, FIXED BIN);

CALL NAM$RM_PORTAL (*entryname, code*);

Parameters

entryname

INPUT. The entry that represents the portal mount point.

code

OUTPUT. The standard return code. Possible values are

E$SCCM	This routine may only be used at the supervisor terminal.
E$NRIT	You do not have access rights for this operation.
E$BNAM	The entryname that you specified uses incorrect syntax.
E$FNTF	The specified portal was not found; delete operation failed.
E$IREM	The specified portal mount must be on a local directory.

Discussion

See also NAM$AD_PORTAL, earlier in this chapter. At Rev. 23.0, the new file system introduces the concept of the **portal**, a new file system object that allows you to reference an object on a remote machine. A portal acts as a gateway between common file system name spaces, thereby allowing you to perform operations upon file system objects in other namespaces, transparently.

For a detailed description of the Rev. 23.0 file system, refer to the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action required.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

PAR$RV

Returns a logical value indicating whether a specified partition supports ACL protection and quotas.

Usage

DCL PAR$RV ENTRY (CHAR (32) VAR, FIXED BIN) RETURNS (FIXED BIN);

par_rev = **PAR$RV (*part_name, code*);**

Parameters

part_name

INPUT. Partition name whose revision number is to be returned. Currently, partition names are 1 – 6 characters long, but space for 32 characters is reserved.

code

OUTPUT. Standard error code. Possible values are

E$FNTF	Partition name not found in disk tables
E$BNAM	Invalid disk partition name

par_rev

RETURNED VALUE. Partition revision number. Possible values are

0	ACLs and quotas not supported
1	Converted to allow ACLs and quotas
–1	Error — see error return code (above)

Discussion

The PAR$RV function call returns a revision stamp whose value depends on whether or not the partition in question allows the use of access control lists (ACLs) for file protection, and quotas for controlling the amount of space allocated to directories contained in the partition. Access control subroutines are described in Chapter 2; quota manipulation subroutines are described later in this chapter. Further information on the use of ACLs and quotas can be found in the *PRIMOS User's Guide.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

PRWF$$

Reads, writes, positions, or truncates a file.

Usage

```
DCL PRWF$$ ENTRY (FIXED BIN, FIXED BIN, PTR OPTIONS
                  (SHORT), FIXED BIN, FIXED BIN (31),
                   FIXED BIN, FIXED BIN);

CALL PRWF$$ (prekey+rwkey+poskey+modekey, funit, LOC(buf), nhw,
             pos, rnhw, code);
```

Or, as a function call:

```
file_position = PRWF$$ (prekey+rwkey+poskey+modekey,
                        funit, LOC(buf), nhw, pos, rnhw, code);
```

Parameters

prekey

INPUT. Indicates the action to be taken before other keys, if any, are executed. Possible value is

K$PEOF	Move the file pointer to the end of the file.

rwkey

INPUT. Indicates the action to be taken. Possible values are

K$READ	Read *nhw* halfwords from the object open on *funit* into *buf*.
K$WRIT	Write *nhw* halfwords from *buf* to the object open on *funit*.
K$POSN	Set the current position to the value in *pos*.
K$TRNC	Truncate the file open on *funit* at the current position.
K$RPOS	Return in *pos* the current position as a number of halfwords from the beginning of the object.

poskey

INPUT. Key indicating the positioning to be performed (if omitted, implies K$PRER). Possible values are

K$PRER Move the file pointer of *funit* the number of halfwords specified by *pos* relative to the current position before performing the action specified by *rwkey*.

K$POSR Move the file pointer of *funit* by the number of halfwords specified by *pos* relative to the position resulting from the action specified by *rwkey*.

K$PREA Move the file pointer of *funit* to the absolute position specified by *pos* before performing the action specified by *rwkey*.

K$POSA Move the file pointer of *funit* to the absolute position specified by *pos* after performing the action specified by *rwkey*.

modekey

INPUT. Key that can be used to transfer all or a convenient (to the system) number of halfwords (if omitted, read or write *nhw*). Possible values are

K$CONV Read or write a convenient number of halfwords (up to the number specified by the parameter *nhw*).

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

funit

INPUT. A file unit number (1 through 15 for PRIMOS II, 1 through 32767 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*)

INPUT/OUTPUT. Pointer to the data buffer to be used for reading or writing. If a buffer is not needed for a given PRWF$$ call, it can be specified as loc(0) in the CALL statement.

nhw

INPUT. The number of halfwords to be read or written (mode=0) or the maximum number of halfwords to be transferred (mode=K$CONV). *nhw* must be between 0 and 65535.

pos

INPUT/OUTPUT. An integer specifying the relative or absolute positioning value depending on the value of *poskey*.

rnhw

OUTPUT. A 16–bit unsigned integer set to the number of halfwords actually transferred when *rwkey* = K$READ or K$WRIT. Other keys leave *rnhw* unmodified.

code

OUTPUT. Standard error code. The following codes are added at Rev. 22.0:

E$ZERO	Indicates that the system found and zeroed out an uninitialized block in a file on a robust partition.
E$IFCB	The disk does not contain enough free contiguous blocks.

file_position

The beginning of the file position where the data transfer occurred.

Discussion

prekey is additive and is executed prior to all other keys. It does not change the behavior of other keys; it merely moves the file pointer to the end of the file, *before* executing the other keys. It thereby allows file extension from a single call to PRWF$$.

prekey also allows applications to save the beginning position of the record just appended to the file. This ensures that the end–of–file position cannot be changed before the new record is appended, thus supporting access integrity with a minimum of file searches by the user application.

pos is always a 32–bit integer. 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, *nhw* and *rnhw* are ignored, and no data is transferred.

A call to read or write *nhw* halfwords causes that number of halfwords 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 points to the end of the transferred data in the file. Using a *poskey* of K$PREA or K$POSA, the user can 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 can move the file pointer backward *pos* halfwords from the current position if *pos* is negative, or forward *pos* halfwords if *pos* is positive. Positioning takes place before or after the data transfer, depending on the key. If *nhw* 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$$ is most frequently used to transfer a specific number of halfwords on a call to PRWF$$. In these cases, the *modekey* is 0 and is normally omitted in PRWF$$ calls. In some cases, such as in a program to copy a file from one file directory to another, a buffer of a certain size is set aside in memory to hold information, and the file is transferred, one bufferfull at a time. In this case, the user normally doesn't care how many halfwords are transferred at each call to PRWF$$, so long as the number of halfwords 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 *nhw* or fewer halfwords in the call to PRWF$$. The number of halfwords transferred is a number convenient to the system, and therefore speeds up program execution. The number of halfwords actually transferred is set in *rnhw*. For examples of PRWF$$ used in a program, refer to the file–manipulation examples in Volume I of this series.

The subkey K$FRCW guarantees that PRWF$$ does not return until the disk record(s) involved are written to disk. The write to disk is performed before executing the next instruction in the program.

Note Since the K$FRCW defeats the disk buffering mechanism, *it should be used with care*; one of its effects is to increase the amount of disk I/O. It should be used only when it is necessary to know that data has been physically written onto a disk (as when implementing error recovery schemes).

Whenever PRWF$$ extends a file physically using the K$FRCW key, PRWF$$ writes the following items to disk for every new block written:

- For a DAM file, the DSKRAT block, the data block, and at least one index block.
- For a CAM file on a standard partition, the extent map, the data block, and at least one DSKRAT block.
- For a CAM file on a robust partition, the extent map and at least one data block. DSKRAT blocks of files on robust partitions are not written to disk.

The E$ZERO error can result from improper shutdown of a disk. When a system halt occurs while a file is being extended logically on a robust partition, some data blocks may not be written to disk. As a result, there may be uninitialized blocks in the file. FIX_DISK -FAST does not detect the uninitialized blocks because it does not check the validity of the data in the file.

When the user tries to read the uninitialized blocks at runtime, PRIMOS determines that the blocks are invalid, zeros them, and returns E$ZERO. E$ZERO is returned only the first time that the block is read or part of the block is written. Using the key K$FRCW prevents a file from having uninitialized blocks.

Whenever PRWF$$ issues a write request to a CAM file for more than one record and the file needs additional space to accommodate the request, PRIMOS allocates space only once, regardless of whether the extent length is set, or the length required is different from that which PRIMOS would allocate by default. For example, if a userrequest requires that 32 records be written to a file that is 4 records long, PRIMOS allocates 32 additional blocks at once. See the section, CAM File Subroutines, later in this chapter for a discussion of extent lengths.

When using the K$FRCW key, the programmer is responsible for ensuring that no other concurrent processes (users) are executing a PRWF$$ call. The file can 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 *rnhw* is set to the number of halfwords actually transferred.

On a `DISK FULL` or `QUOTA EXCEEDED` error, the file pointer is set to the value it had at the beginning of the call to PRWF$$. The user can, therefore, delete another file and restart the program (by typing START after using the DELETE command).

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 halfword that is read is the first one of the file. The file pointer value is 0, for the beginning of file. If the user calls PRWF$$ to read 490 halfwords, and does no positioning at the end of the read operation, the file pointer is set to 490.

Note In V–mode, PRWF$$ transfers words only into and out of the same segment as that containing the beginning of the buffer. Reading across a segment boundary causes a wraparound, and reads into the beginning of the segment. Wraparound can also occur when writing from the buffer.

The following examples show some uses of the PRWF$$ subroutine call.

Example 1: Read the next 79 halfwords from the file open on unit 1:

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

Example 2: Add 1024 halfwords to the end of the file open on UNIT (10000000 is just a very large number to get to the end of the file; NMW holds the number of halfwords actually written):

```
CALL PRWF$$ (K$POSN+K$PREA, UNIT, LOC(0), 0, 10000000, 0,
             CODE)

CALL PRWF$$ (K$WRIT, UNIT,  LOC(BFR),  1024,  000000,
             NMW, CODE)
```

Example 3: See what position is on file unit 15 (INT4 is INTEGER*4):

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

Example 4: Truncate file 10 halfwords beyond the position returned by the above call:

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

Example 5: Position the file open on unit number UNIT to the tenth halfword used in the file; then write the first 10 halfwords of ARRAY 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 causes the file that is open on unit number UNIT to be positioned to the tenth halfword in the file, and the first 10 halfwords of ARRAY are written to it. The next instruction in the user's program is not 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 is not performed immediately.

Example 6: Read and write SAM and DAM files using PRWF$$:

```
/******************************************************************
/* Copy SAM, DAM, or CAM 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 of open source file  */
     tunit         fixed binary(15), /* unit of target file       */
     err_info      fixed binary(15), /* if code ^= 0 indicates    */
                                     /* file that 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 prwf$$ read  */
dcl  words_written fixed binary(15); /* actual words prwf$$ wrote */
dcl  eof           bit(1);
dcl  recbuf_ptr    pointer options(short);
dcl  addr          builtin;
dcl  er$print      entry(fixed bin(15, char(*) var, fixed bin(15),
                   char(*) var, char(*) var);
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 val           */
                   fixed binary(15),  /* actual words read or wrtn*/
                   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$dkfl
            then do;
             call er$print(k$irtn, sscs$errd, code, '', 'cp$$fl');
              call user_proc;     /* Wait for response */
                   go to a;
                   end;
            else do;
                   err_info = 2;
                   return;
                   end;
end;
return;
end cp$$fl;

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

More examples of the use of PRWF$$ are given with the file system examples in the *Subroutines Reference I: Using Subroutines.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

Q$READ

Returns directory quota and disk record use information.

Usage

DCL Q$READ ENTRY (CHAR(128)VAR, (8) FIXED BIN (31),
FIXED BIN, FIXED BIN, FIXED BIN);

CALL Q$READ (*pathname, quota_info, max_entries, type, code*);

Parameters

pathname

INPUT. Name of the directory whose quota information is to be read. 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

OUTPUT. Structure in which quota information is returned. Format is described below.

max_entries

INPUT. Number of entries in *quota_info*. Maximum is six for Rev. 20.2 and later revisions.

type

OUTPUT. Type of directory (input). Possible values are

0	Quota Directory
1	Nonquota Directory

code

OUTPUT. Standard error code. Possible value is

E$NINF	Insufficient access to read quota.

Discussion

Quota and disk use accounting concepts are explained in the *System Administrator's Guide, Volume I: System Configuration.*

The Q$READ subroutine returns a maximum of six items of information. If more than six are requested, six are returned. A user program can specify, in *max_entries*, a smaller number of items; if a value n ($1 < n < 6$) is specified, the first n items of the structure are returned. The contents of the two reserved entries are undefined at Rev. 20.2. The user declares the structure as follows:

```
DCL 1 quota_info,
      2 record_size FIXED BIN (31),
      2 dir_used FIXED BIN (31),
      2 max_quota FIXED BIN (31),
      2 quota_used FIXED BIN (31),
      2 rec_time_product FIXED BIN (31),
      2 dtm FIXED BIN (31),
      2 reserved_1 FIXED BIN (31),
      2 reserved_2 FIXED BIN (31);
```

record_size

Record size in halfwords: 440 or 1024.

dir_used

Records used in this directory.

max_quota

Quota for this directory.

quota_used

Records used in subtree of this directory.

rec_time_product

Cumulative record-minutes for this directory.

dtm

Date/time when *rec_time_product* was last updated, in standard file–system date format. (See Appendix C of Volume III for more information about this format.) The *dtm* value is always the time when Q$READ is executed, because execution of Q$READ forces the quota block to be flushed to disk.

When this call is invoked on a nonquota directory, *type* has a returned value of 1, and *max_quota*, *rec_time_product*, and *dtm* have returned values of 0. The value returned in *dir_used* indicates the sum of the records used in the files in the directory and the records used by the directory itself. *quota_used* indicates the sum of the records used for all files and subdirectories of this directory.

Quota directories return a *type* value of 0, and all requested quota information.

The system keeps an accounting use 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 32–bit integer, which is to say that it counts to all ones (some 4.3 billion) and then goes to 0. The system also indicates when the last update occurred.

The USAGE is computed in record–minutes, computed according to the formula:

TIME = (Current date/time) – (Date/time quota last modified)

USAGE = USAGE + (*quota_used*) * TIME

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

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

Q$SET

Sets a quota on a subdirectory in the current directory.

Usage

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

CALL Q$SET (***key, pathnam, max_quota, code***);

Parameters

key

INPUT. Must be K$SMAX (set maximum quota).

pathname

INPUT. An array containing the name of the subdirectory to receive the quota.

max_quota

INPUT. Maximum quota for the directory and its subtree.

code

OUTPUT. Standard return code. Possible values are

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.

Discussion

If the directory specified in *pathname* is not already a quota directory, it is converted to a quota directory.

The user must have Protect access to the directory's parent.

If *max_quota* is specified as 0, any quota already existing on the directory is removed, and the directory becomes a nonquota directory. If *max_quota* is assigned a value that is less than the number of records already used in this directory, a warning is returned, but the quota is set to the new value. Under these conditions, the user will receive a `MAXIMUM QUOTA EXCEEDED` message whevever an attempt is made to add records to a file in the directory. The number of records used in the directory must be reduced (normally by deleting old or unneeded files) to less than the value of the new quota.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

RDEN$$

RDEN$$ positions in or reads from a directory.

This subroutine is considered obsolete, and its use in new programming is discouraged. Use DIR$RD or ENT$RD instead. Users maintaining existing programs that call RDEN$$ can refer to Appendix A for a complete description of the subroutine.

RDLIN$

Reads a line of characters from an ASCII disk file.

Usage

DCL RDLIN$ ENTRY (FIXED BIN, CHAR(*), FIXED BIN,
FIXED BIN)[, RETURNS (BIT(16))];

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

Or, as a function call:

line_length **= RDLIN$ *(funit, buffer, count, code);***

Parameters

funit

INPUT. File unit on which the file to be read is open.

buffer

INPUT. Name of a varying string of *count* halfwords in which the line of information from the disk file is to be read.

count

INPUT. Size of *buffer* in halfwords.

code

OUTPUT. Standard error code. The following code is added at Rev. 22.0:

E$ZERO	Indicates that the system found and zeroed out an uninitialized block in a file on a robust partition.

line_length

OPTIONAL RETURNED VALUE. The length of the line just read, which may be one of the following:

–1	End of file
0	Any error condition

Discussion

A line of characters from the file open on *funit* is read into the area specified by *buffer*, two characters per halfword. Lines on the disk are separated by the newline character. For compressed files, when a control character DC1 (221 octal) followed by a number is read from the disk, the DC1 is suppressed and the number is replaced by that many spaces in the buffer.

If the line being read is less than twice the *count* characters, the remaining characters in the buffer are filled with spaces. If it is greater than twice the *count* characters, only twice the *count* characters fill the buffer and the remaining characters on the disk file line are lost. The newline character itself never appears as part of the line read into the buffer.

When RDLIN$ detects an uninitialized block, it returns any valid data. Whenever the start of a line is lost, it is not possible to determine where the end–of–line is. For this reason, whenever RDLIN$ encounters an uninitialized block, the position in the file is set to the beginning of this block, and the operation is terminated. If the read request spans blocks, whatever data is available from valid blocks is returned in the buffer.

The E$ZERO error can result from improper shutdown of a disk. When a system halt occurs while a file is being extended logically on a robust partition, not all data blocks may be written to disk. As a result, there may be uninitialized blocks in the file. FIX_DISK –FAST will not detect the uninitialized blocks because it does not check the validity of the data in the file. When the user tries to read the uninitialized blocks at runtime, PRIMOS determines that the blocks are invalid, zeros them, and returns E$ZERO. E$ZERO is returned only the first time that the block is read or part of the block is written.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

SATR$$

Sets or modifies an object's attributes in its directory entry.

Usage

```
DCL SATR$$ ENTRY (FIXED BIN, CHAR (32), FIXED BIN,
                  (2) FIXED BIN, FIXED BIN);

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

Parameters

key

INPUT. Specifies the action to take. Possible values are

K$PROT Set password protection attributes from the first halfword of the *attributes* array. The second halfword of *attributes* is ignored for pre–Rev 19.0 partitions and must be 0 for Rev 19.0 and newer partitions.

K$DTIM Set date/time modified from both halfwords of *attributes*.

K$DTB Set date/time backed up from both halfwords of *attributes*.

K$DTC Set date/time created from both halfwords of *attributes*. Note that only the system console (user 1), or users who belong to the .BACKUP$ ACL group, can use K$DTC.

K$DTA Set date/time last accessed from both halfwords of *attributes*. Note that only the system console (user 1), or users who belong to the .BACKUP$ ACL group, can use K$DTA.

K$DMPB Set the dumped bit. This bit is set by the utility program that takes backup dumps of modified files, and is reset by the operating system whenever the file is modified.

Caution It is important to use the K$DMPB key with care, because indiscriminate resetting of the dumped bit can result in failure to back up the affected file.

K$RWLK Set the read/write lock on a per–file basis. Bits 15 and 16 of the first halfword of *attributes* are set by the user for specific lock values.

K$SDL Set the delete switch (for use with ACLs). If the first halfword of *attributes* is not 0, the delete switch is set. If it is 0, the switch is cleared.

K$LTYP Set the logical type field in the file entry to the value in the first word of *attributes*. This field should never be set by user software. It is for Prime internal use only.

K$TRUN Set the "truncated by FIX_DISK" bit from the value in bit 1 of the first halfword of *attributes*. This field should never be set by user software. It is for Prime internal use only.

object

INPUT. Name of the object whose attributes are to be modified. The current directory is searched for *object*.

namlen

INPUT. Length in characters of *object*.

attributes

INPUT. Field containing the attributes; one or two halfwords, depending on *key*. Possible values are

K$PROT A 16–bit (one–halfword) structure defining the password protection rights for the object, as defined below.

K$DTxx A 32–bit (two–halfword) structure containing the date/time to set, in standard FS format.

K$DMPB Ignored.

K$RWLK One of the following sub–keys:

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

K$SDL A 16–bit (one–halfword) quantity. If nonzero, the delete–protect switch is set on. If zero, it is set off.

code

OUTPUT. Standard error code. Possible values are

E$BKEY An invalid key value was passed.

E$BNAM Object name is invalid.

E$BPAR *namlen* is less than 1 or greater than 32.

E$NATT The current attach point is invalid.

E$NRIT Protect access (Delete access for K$SDL) is missing from the current directory; or, a user who does not belong to the .BACKUP$ ACL group has attempted to use SATR$$ with K$DTA or K$DTC.

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.

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

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

E$ATNS The attribute is not supported in the directory, which is in pre–Rev. 20.2 format.

Discussion

The attributes that can be set include

- Password protection
- Date/time modified, backed up, created, or accessed
- Dumped bit
- Read/write lock
- Delete–protect switch

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 is structured as described in Appendix C of Volume III.

Note SATR$$ does not check the validity of the supplied date and time. Users must assure that the date/time passed is legal.

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

Since a call to SATR$$ modifies the directory, the date/time modified and date/time last accessed of the directory itself are updated.

The PRIMOS file system supports read/write locking (concurrency) on a per–file basis. The read/write lock is used to regulate concurrent access to the file, and was formerly alterable only on a systemwide basis. The read/write lock bits are bits 5 and 6 of*file_info*, as described for the DIR$LS subroutine, earlier in this chapter.

The meaning of the lock values is

Value	*Bits 5, 6*	*Meaning*
0	0, 0	Use systemwide 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.

Files are created with read/write lock bits set to 00.

User directories do not have user–alterable read/write locks, although segment directories do. Files in a segment directory have the per–file read/write lock of the segment directory.

The per–file read/write lock value can be read by any of the directory reading subroutines: DIRLS, DIRRD, DIR$SE, or ENT$RD. It is set by a SATR$$ call with a key of K$RWLK. The desired value is supplied in bits 15 and 16 of the first halfword of *attributes*, the remaining bits of which must be 0. On pre–Rev. 19.0 partitions, the SATR$$ call fails with an error code of E$OLDP. Owner rights to the containing directory are required, otherwise the call fails with an error code of E$NRIT.

An attempt to set the lock value of a directory fails with an error code of E$DIRE. If the SATR$$ call requests a lock value which is more restrictive than the current use of the file, the file's lock value is changed and current users of the file are unaffected, but any subsequent open requests are governed by the new lock value.

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 systemwide 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.

Owner rights are required on the directory 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 a pre–Rev. 19.0 partition results in an E$OLDP error.

The following examples illustrate some uses of the SATR$$ subroutine.

Example 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)
```

Example 2: Set both owner and nonowner attributes to read–only (note carefully the bit positioning in two–halfword octal constant):

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

Example 3: Set date/time modified from directory entry read into ENTRY by RDEN$$:

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

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

SGD$DL

Deletes a segment directory entry.

Usage

DCL SGD$DL ENTRY (FIXED BIN, FIXED BIN);

CALL SGD$DL (*segdir_unit, code*);

Parameters

segdir_unit

INPUT. Unit on which the segment directory is open.

code

OUTPUT. Standard error code. Possible values are

E$BUNT	*segdir_unit* contains an invalid value.
E$SUNO	Unit is not open, or is not open for writing.
E$NTSD	Object open on *segdir_unit* is not a segment directory.
E$FNTS	Entry at the current position does not exist, or the segment directory is 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 call such as SRCH$$), and must be positioned (by an SGDR$$ call) at the entry to be deleted.

Delete access is required to the segment directory containing the member to be deleted. The date/time modified and date/time accessed fields are updated in the segment directory.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SGD$EX

Finds out whether there is a valid entry at the current position within the segment directory open on a specified unit.

Usage

DCL SGD$EX ENTRY (FIXED BIN, FIXED BIN, FIXED BIN);

CALL SGD$EX (***unit, type, code***);

Parameters

unit

INPUT. Specifies the unit number on which the segment directory is open.

type

OUTPUT. Type of file detected (SAM or DAM).

code

OUTPUT. Standard error code.

Discussion

SGD$EX attempts to read the entry at the current position. If there is no valid SEGDIR entry at that position, SGD$EX returns the error E$FNTS (`NOT FOUND IN SEGMENT DIRECTORY`).

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SGD$OP

Opens a segment directory entry.

Usage

```
DCL SGD$OP ENTRY (FIXED BIN, FIXED BIN, FIXED BIN,
                  FIXED BIN, FIXED BIN, FIXED BIN)
                  RETURNS (FIXED BIN);
```

open_unit = SGD$OP (*key, seg_unit, file_unit, file_type, new_type, code*);

Parameters

key

INPUT. Mode in which object is to be opened. Possible values are

K$READ	Open object for reading (input only).
K$WRIT	Open object for writing (output only).
K$RDWR	Open object for reading and writing (input/output).
K$VMR	Open object for virtual memory file access (VMFA) reading. Used only before calling one of the EPF subroutines for initializing or executing an EPF.

seg_unit

INPUT. File unit on which the segment directory containing the entry is opened. The segment directory must have been opened (by a call to SRCH$$, for example) before the call to SGD$OP can be issued.

file_unit

INPUT. File unit on which the entry is to be opened. Supply either a specific file unit number between 1 and 126, or the value –10000 to cause PRIMOS to select one. The selected unit number is returned in *open_unit*.

file_type

OUTPUT. Type of file opened. Possible values are

0	Sequential access (SAM) file
1	Direct access (DAM) file
2	Sequential access segment directory (SEGSAM)
3	Direct access segment directory (SEGDAM)
7	Contiguous access (CAM) file

new_type

INPUT. Type of object to be created if it does not exist (*key* must be K$WRIT or K$RDWR). Possible values are

0	Create a sequential access (SAM) file.
1	Create a direct access (DAM) file.
2	Create a sequential access segment directory (SAM Segdir).
3	Create a direct access segment directory (DAM Segdir).
7	Create a contiguous access (CAM) file.

code

OUTPUT. Standard error code.

open_unit

RETURNED VALUE. File unit number of the newly opened entry.

Discussion

A full description of the SGD$OP subroutine is given in the *Advanced Programmer's Guide II: File System.*

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SGDR$$

Positions in, reads an entry in, or modifies the size of a segment directory.

Usage

DCL SGDR$$ ENTRY (FIXED BIN, FIXED BIN, FIXED BIN,
FIXED BIN, FIXED BIN);

CALL SGDR$$ (***akey+filekey, funit, entrya, entryb, code***);

Parameters

akey

INPUT. Key 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*. The directory must be open for reading or for both reading and writing. One of the following values is returned in *entryb*:

- 1 If the position given by *entrya* exists and contains a file.
- 0 If the position given by *entrya* exists but does not contain a file.
- –1 If the position given by *entrya* is beyond the end of the directory (EOD).

If EOD is reached on K$SPOS, the file pointer is left at EOD.

K$FULL — Move the file pointer of *funit* to the position given by the value of *entrya*. One of the following values is returned in *entryb*:

- 1 The position given by *entrya* if this position is full.
- 0 The position of the next full entry if the position at *entrya* is empty.
- –1 If the position at *entrya* is empty and there are no full positions beyond it. The file pointer is left set at EOD.

K$FREE — Same as for K$FULL, but find an entry that does *not* contain a file.

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

K$GPOS Return in *entryb* the file entry number currently 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 directory. The directory must be open for both reading and writing.

K$MVNT Move the entry pointed to by *entrya* to the entry pointed to by *entryb*. The *entrya* entry is replaced with a null pointer. An error is returned by K$MVNT if there is no file at *entrya*, if there is already a file at *entryb*, or if either *entrya* or *entryb* is at or beyond the end of the directory. The file pointer is left at an undefined position. The directory must be open for both reading and writing.

K$FACR Locate a free entry in a segment directory and return the assigned entry number to *entryb*. If *akey* equals K$FACR, you must select the type of file by specifying the *filekey* attribute.

filekey

INPUT. The type of subfile to be created if *akey* equals K$FACR. Possible values are

K$NSAM	New SAM file
K$NDAM	New DAM file
K$NSGS	New SAM segment directory
K$NSGD	New DAM segment directory
K$NCAM	New CAM file

funit

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

entrya

INPUT. Entry number in the directory, to be interpreted according to value of *key*.

entryb

INPUT/OUTPUT. Integer set or used according to value of *key*.

code

OUTPUT. Standard error code. Value returned depends on value of *key*.

Discussion

When SGDR$$ is called, the segment directory must not be opened for Write–only access. Whether Read–only or Read and Write access is required depends on the action to be performed, as determined by the value of *key*.

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 using the K$POS key with *entrya* values of *n*, *n+1*, *n*+2,..., the end of the directory is indicated by returning a –1 in *entryb*, rather than by returning the E$EOF error code. E$EOF is returned when *entrya* reaches a value greater than the value that returned –1 in *entryb.*

The following examples illustrate some uses of the SGDR$$ call.

Example 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
```

Example 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)
```

Example 3: Read and write 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,
     tunit      fixed bin,
     err_info   fixed bin,
     code       fixed bin;
dcl (entrya,
     entryb,
     entry_no)  fixed bin;
dcl (sfunit,
     tfunit)    fixed bin;
dcl (newfil,
     trash,
     tcode,
     rtnval,
     type)      fixed bin;
dcl  er$print   entry(bin, char(*) var, bin, char(*) var,
                      char(*) var);
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 (fixed bin, /* read segdir entries */
                              /* first is key */
                fixed bin,    /* unit on which segdir is open */
                fixed bin,    /* entrya */
                fixed bin,    /* entryb */
                fixed bin);   /* 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$spos, tunit, entry_no, entryb, code);
     if code ^= 0
        then go to err_rtn_2;
     if entryb < 0
        then do;
             call er$print(k$irtn, ssc$errd, e$null,
                          'Unrecoverable error','cp$$sd');
             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;
```

```
/* 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;
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

SIZE$

Returns the size of a file system entry.

Usage

```
DCL SIZE$ ENTRY (CHAR(128) VAR, FIXED BIN(15),
                 FIXED BIN(31), PTR, FIXED BIN(15),
                 FIXED BIN(15));
```

CALL SIZE$ (*pathname, expected_version, rec_size, buf_ptr, buf_size, code*);

Parameters

pathname

INPUT. Pathname of the entry whose size is desired.

expected_version

INPUT. Version of output structure expected by caller. Must be 1.

rec_size

INPUT. Record size in halfwords. This is used for calculating the file size in records. It should be set to 1 if the output units desired are in halfwords.

buf_ptr

INPUT -> OUTPUT. Pointer to the caller's buffer.

buf_size

INPUT. Size of caller's buffer in halfwords.

code

OUTPUT. Standard error code. Possible values are

E$BFTS	Buffer too small (returned only if *buf_size* < 1).
E$FNTF	Entry does not exist.
E$NRIT	Insufficient access rights.
E$BVER	Invalid version number.
E$BPAR	Bad parameter (*rec_size* < 1).

Discussion

The SIZE$ subroutine returns the size (in halfwords) and type of the object specified by *pathname*. If the object is one that contains subentries (a file directory, a segment directory, or an access category), the number of subentries is also returned. For directories, SIZE$ indicates whether or not the object is an ACL directory.

The caller must have Read access if the object is a file or a segment directory, or List access if it is a file directory. List and Use rights to the parent directory are also required.

SIZE$ does not alter the date/time accessed (DTA) field of the specified object. It does, however, modify the DTA of the parent directory.

If the buffer size specified in the *buf_size* parameter is too small for the entire structure, the first *buf_size* halfwords are returned.

The following is the structure returned by the subroutine in the caller's buffer:

```
DCL 1 size_info,
      2 version_number FIXED BIN(15),/* Must be 1 */
      2 struc_len FIXED BIN(15),    /* Structure size (bytes) */
      2 entry_type FIXED BIN(15),   /* Type, as follows:
                                    0:  SAM file
                                    1:  DAM file
                                    2:  SAM segment dir
                                    3:  DAM segment dir
                                    4:  User directory
                                    6:  ACAT
                                    7:  CAM file */
      2 logical_size FIXED BIN(31), /* Size in halfwords divided
                                    by record_size. For SD,
                                    total size of member
                                    files. 0 for ACATs. */
      2 phys_recs FIXED BIN(31),    /* Valid only for CAM files.
                                    No. of physical records.*/
      2 is_acl_dir BIT(1) ALIGNED,  /* Valid only for user
                                    directory
                                    1' b -> ACL directory.   */
      2 num_entries FIXED BIN(31),   /* Valid only if entry is an
                                    ACAT, user directory, or
                                    SD. Total number of
                                    entries in ACAT or user
                                    directory, maximum number
                                    of entries for SD.      */
      2 num_full_entries FIXED BIN(31); /* Valid only for SD.
                                    Current number of full
                                    entries.    */
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SRCH$$

Opens, closes, deletes, changes access, or verifies the existence of an object.

Usage

```
DCL SRCH$$ ENTRY (FIXED BIN, CHAR (32), FIXED BIN,
                   FIXED BIN, FIXED BIN, FIXED BIN);

CALL SRCH$$ (action+ref+newfil, object_name, nam_len, funit, type,
               code);
```

Parameters

action

INPUT. Indicates the action to be performed. Possible values are

K$READ — Open *object_name* for reading on *funit.*

K$WRIT — Open *object_name* for writing on *funit.*

K$RDWR — Open *object_name* for reading and writing on *funit.*

K$CLOS — Close *object_name* or object open on *funit.*

K$DELE — Delete *object_name.*

K$EXST — Verify existence of *object_name.*

K$VMR — Open *object_name* for VMFA read. Valid only if *object_name* is an EPF–format runfile.

ref

INPUT. Modify the *action* key as follows:

K$IUFD — Search for *object_name* in the current directory. (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 specified in *object_name.*

K$CACC — Change the access mode of the file already open on *funit* to that specified in *action* (K$READ, K$WRIT, or K$RDWR only).

K$GETU — Open *object_name* on an unused file unit selected by PRIMOS. The unit number is returned in *funit.* See Example 6 for use of this key.

K$CURR	Open a unit on the current attach unit.
K$NMNT	If *object_name* is a disk/portal mount point, the error code E$MTPT is returned.

newfil

INPUT. Type of file to create if *object_name* does not exist and *action* is K$WRIT or K$RDWR. Possible values are

K$NSAM	New SAM file (the default)
K$NDAM	New DAM file
K$NSGS	New SAM segment directory
K$NSGD	New DAM segment directory
K$NCAM	New CAM file

Note It is not possible to create a directory with SRCH$$; use DIR$CR instead.

object_name

INPUT. Name of the object to be opened (1 - 32 characters). SRCH$$ reads this string until it encounters a blank, reads the number of characters specified in *nam_len*, or reads 32 characters. Therefore, a string of blanks or an *object_name* beginning with a blank is treated as a null string; a null string defaults to the name of the current directory. An *object_name* of K$CURR can also be used to open the current directory (*action* keys K$READ, K$WRIT, or K$RDWR only).

If *ref* is K$ISEG, *object_name* is a file unit from 1 through 126 (1 through 15 under PRIMOS II) on which a segment directory is already open.

nam_len

INPUT. Length in characters (1–32) of *object_name.*

funit

INPUT/OUTPUT. Number of the file unit to be opened or closed (input). When SRCH$$ is used with *ref* = K$GETU, *funit* returns the PRIMOS–selected file unit number selected by PRIMOS.

type

OUTPUT. 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 of *type* are

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	User directory
7	CAM file

code

OUTPUT. Standard error code. The following error codes are specific to this subroutine:

Keyword	*Value*	*Meaning*
E$NATT	7	No root–entry directory attached. This error usually occurs only when the directory to which the user is attached is removed from the system, as when a disk is shut down.
E$NRIT	10	Insufficient access rights. You do not have List access to the current directory.
E$MTPT		Directory is a mount point. You cannot reference a directory that is the mount point of another directory, so that tape backup procedures do not cross mount points. This error is returned by the *ref* key K$NMNT.

Discussion

The SRCH$$ subroutine has multiple uses. The most common use is to open and close files. It can also be used to add, delete, change access to, and verify the existence of file system objects.

Note The delete functions of SRCH$$ are better performed by FIL$DL and SGD$DL.

Opening Objects: Opening an object consists of connecting the object to a file unit. After an object is opened, various input, output, and positioning actions can be performed on it. These actions are accomplished by other subroutines, which reference the object through the associated file unit: PRWF$$, SGDR$$, RDEN$$, RDLIN$, WTLIN$, I$AD07, O$AD07, RDASC, and WRASC. Information can also be transferred through I/O statements in all high–level languages.

On opening an object, SRCH$$ specifies

- Operations that can be performed by other subroutines. These operations are read–only, write–only, or both read and write.
- Where to look for the object, or where to add the object if it does not currently exist. SRCH$$ specifies either the name of an object in the currently attached directory or a file unit number on which a segment directory is open. In the segment directory reference, file unit's current position pointer indicates the segment directory member to be opened.

For an ACL–protected object, the user must have access to the object and its containing directory appropriate to the action to be performed; the object's access control list specifies thc rights a given user or group has to the object.

For password–protected objects, each object in a directory has two sets of access rights, one for the owner and one for the nonowner of the directory. When an object is created, its owner has all rights (Read, Write, Delete), and nonowners have none. These rights can be changed using the PROTECT command or the SATR$$ subroutine. The access rights are checked on any attempt to open an object. SRCH$$ returns a NO RIGHTS error code (E$NRIT) if the user does not have the required rights under either kind of protection.

If a file cannot be found when opening for reading, SRCH$$ returns the FILE NOT FOUND error code (E$FNTF). If the file unit is already in use, SRCH$$ generates the unit–in–use error code (E$UIUS).

Closing an Object: The SRCH$$ subroutine can close an object by name or by file unit. SRCH$$ attempts to close by *object_name* unless *nam_len* is specified as 0, in which case it closes the file unit specified. If *object_name* 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 does not return an error code (unless *funit* is out of range).

If the disk is not write–protected, closing the object updates its date/time last accessed field. If the object was modified while it was open, closing it updates its date/time modified field as well.

The Read/Write Lock: By default, PRIMOS allows any number of readers, or a single writer and no readers for the same object. The system prevents one user from opening a file for writing when another user has the file open for reading or writing. It also 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 more than one file unit. If a lock violation is attempted, SRCH$$ returns the FILE IN USE (E$FIUS) error code.

This lock can be changed on a per–file basis. (Refer to the SATR$$ subroutine, described earlier in this chapter.)

Changing the Access Mode of an Open Object: Using the K$CACC subkey, a user can change the access mode of an object that is open on *funit* to open for reading, open for writing, or open for both reading and writing. Note that access rights and the read/write lock rules for the object are checked and the attempt to change access may fail.

Adding Objects in Directories: A call to SRCH$$ to open a file for writing or both reading and writing causes SRCH$$ to look in the current directory for the file. If it is not found in the directory, SRCH$$ creates a new file of zero length and puts an entry for the file into the directory.

The date/time created and the date/time accessed fields of the file are set to the current date/time, the access rights are set to their default values, the read/write lock is set to the system default, and the file type to the type specified by the *newfil* subkey. If the *newfil* subkey is not specified, it is a SAM file.

On a robust partition, SRCH$$ creates the file as a CAM file and returns a value of 7 (= CAM file) to *type*, regardless of the file type requested in *newfil.* CAM files are created with one block of physical space.

Verifying the Existence of a File: The K$EXST key can be used to determine whether a specific object exists in the current directory or in a segment directory. The object is not affected in any way. The access rights and the read/write lock are not checked, nor is the date/time last accessed field changed. If an illegal filename is given, SRCH$$ returns the status code E$BNAM.

Operations on Subdirectories: The contents of entries of subdirectories can be read through calls to ENTRD, DIRLS, DIRRD, DIRSE, and GPAS$$ once the subdirectory is open. The current directory can be opened by specifying the key K$CURR in the *object_name* field of the SRCH$$ call. While the current directory can be opened for writing, or for reading and writing, write operations such as PRWF$$ cannot explicitly write to the directory. Only implicit writes, such as those performed automatically when updating the directory to reflect changes, are permitted.

Calls to the SATR$$ or SPAS$$ subroutines require that the current directory not be open; otherwise, the `FILE IN USE` error is returned. New directories can be created only by using the CREA$$ subroutine; SRCH$$ does not allow creation of a directory. Directories can be deleted with SRCH$$ only if the directory contains no files. The DELETE command can delete a nested structure of directories, provided they are not protected.

Operations Involving Segment Directories: Segment directories are directories in which the files are referenced numerically 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, and the read/write lock, of the highest level segment directory in the directory. Segment directories are not attached to, but are operated on using SRCH$$ and SGDR$$.

To create a segment directory, use SRCH$$ to open a new object for reading and writing with *newfil* specified as K$NSGS or K$NSGD.

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

To create a file in a segment directory, follow these steps:

1. Open the directory for reading and writing on a file unit (for example, SUNIT), if it is not already open.
2. Use SGDR$$ to position to the null file entry into which the new file is to be placed.
3. Use SRCH$$ to open a new file in the segment directory for writing, or for reading and writing. Use the K$ISEG reference key and place the SUNIT number of the segment directory in the *object_name* parameter. Place the file unit of the new file in the *funit* parameter. SRCH$$ creates the new file and places a pointer to the new file in the segment directory entry specified by SUNIT.

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

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 is opened. If not, and the attempt is to open for reading, the `FILE NOT FOUND` error is returned. Any object type except a directory can 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$$ with the K$ISEG and K$DELE subkeys. SRCH$$ returns the object's records 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, first delete all files in the directory using SGD$DL, set the size of the directory to 0 using SGDR$$, close the directory, and then delete it with FIL$DL. The DELETE subcommand of the SEG command can also be used to delete a segment directory.

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

The following examples illustrate some uses of the SRCH$$ subroutine.

Example 1: Open new SAM file, RESULTS, for output on file unit 2:

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

Example 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)
```

Example 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)
```

Example 4: See if filename MY.BLACK.HEN is in current directory:

```
CALL SRCH$$ (K$EXST+K$IUFD, 'MY.BLACK.HEN', 12, 0, TYPE,
             CODE)
IF (CODE.EQ.E$FNTF) CALL TNOU('NOT FOUND', 9)
```

Example 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)
```

Example 6: Open the file named FILE in the user's currently attached directory:

```
CALL SRCH$$(K$READ+K$GETU, 'FILE', 4, UNIT, TYPE, CODE)
IF (CODE .NE. 0) GOTO error_processor
```

The above FORTRAN call attempts to open the file named FILE in the user's current directory. If successful, the file unit number on which FILE is opened is returned in UNIT, the type of the file opened is returned in TYPE, and CODE is set to 0. If there are any errors, CODE is 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.

Example 7: Open file by name:

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

open$:
     proc(key, fullname, 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,
   fullname       char(*) var,
   treename       char(128) var,
   treelength     bin,
   unit           bin,
   type           bin,
   code           bin;
dcl  srch$$       entry(bin, char(*), bin, bin, bin, bin),
     newfil       bin;
dcl  at$          entry(bin, char(128) var, bin);
dcl  at$hom       entry(bin);
dcl  extr$a       entry(char(*) var, char(*) var, bin,
                  char(32) var, bin);
dcl  full         bit(1) aligned;
dcl  tree         bit(1) aligned,
     filename     char(32) var;
dcl  length       bin;

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

     code = 0;
     full = (index(fullname, '>') ^= 0);
     if full
        then do;
    call extr$a(fullname,treename,treelength,filename,code);
     if code ^= 0 then go to clean_up;
     tree = (index(treename, '>') ^= 0);
     if full | tree
        then do;
             call at$ (k$setc, treename, code);
            if code ^= 0
                then go to clean_up;
             end;
         end;
```

```
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,
                 unit, type, code);

clean_up:
     if tree
        then call at$hom (code);
     return;

     end open$;
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

SRSFX$

Searches for a file with 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$ (action+null_suffix+ref+newfile, object_name, funit, type,
             n_suffixes, suffix_list, basename, suffix_used, code);
(or)
chrpos = SRSFX$ (action+null_suffix+ref+newfile, object_name, funit,
             type, n_suffixes, suffix_list, basename, suffix_used, code);
```

Parameters

action

INPUT. Action to be performed. Possible values are

K$READ	Open *object_name* for reading on *funit.*
K$WRIT	Open *object_name* for writing on *funit.*
K$RDWR	Open *object_name* for reading and writing on *funit.*
K$CLOS	Close *object_name.*
K$DELE	Delete *object_name.*
K$EXST	Check on existence of *object_name.*
K$VMR	Open *object_name* for VMFA read. Valid only if *object_name* is an EPF–format runfile.

null_suffix

To search for a file with no suffix first, use a plus sign to concatenate K$NULF to the *action* argument (for example, K$WRIT+K$NULF). *null_suffix* is an optional argument. If you specify K$NULF, PRIMOS first searches for *object_name* with no suffix, then searches for *object_name* with the suffixes specified in *suffix_list.* If you do not specify K$NULF, PRIMOS searches for *object_name* with no suffix last.

ref

INPUT. Modifies the *action* key as follows:

K$IUFD Search for *object_name* in the current directory. (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 specified in *object_name.*

K$CACC Change the access mode of the file already open on *funit* to that specified in *action* (K$READ, K$WRIT, or K$RDWR only).

K$GETU Open *object_name* on an unused file unit selected by PRIMOS. The unit number is returned in *funit.*

newfile

INPUT. Indicates the type of file to create if *object_name* does not exist and *action* is K$WRIT or K$RDWR. Possible values are

K$NSAM	New SAM file (the default)
K$NDAM	New DAM file
K$NSGS	New SAM segment directory
K$NSGD	New DAM segment directory
K$NCAM	New CAM file

object_name

INPUT. Pathname to use for search. A null string ( ) or a string of blanks refers to the current directory.

funit

INPUT. File unit opened (returned with K$GETU) or file unit to use for SRCH$$ action without K$GETU.

type

OUTPUT. File type opened. Note that for CAM files, the *type* value returned is 7, even though the value of K$NCAM is 4.

n_suffixes

INPUT. Number of suffixes in *suffix_list.* A value of 0 indicates not to use the file naming standards with suffixes for the search.

suffix_list

INPUT. List of desired suffixes to use. Each suffix should include the period and be in capital letters, for example, *suffix_list(i)*'='.F77. The suffixes can have varying lengths; therefore the suffix list of variable strings is declared as (*)CHAR(32)VAR.

basename

OUTPUT. Base filename (that is, without a suffix) to be searched for according to the suffix list.

suffix_used

OUTPUT. Index, in the suffix list given, of the suffix used for the search. A value of 0 denotes that the null suffix was used.

code

OUTPUT. Standard error code.

chrpos

OPTIONAL RETURNED VALUE. When SRSFX$ is called as a function, a FIXED BIN(31) value is returned. The first halfword points, in the case of an invalid pathname, one character past the pathname component that caused the error. The second halfword is the pathname length.

Discussion

SRSFX$ is intended for use with the file naming convention 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 suffixes in the list are found appended to *pathname*, 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 subroutine returns when a filename suffix is found or the suffix list is exhausted. When the subroutine returns, it reattaches to the home directory.

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.

SRSFX$ can be combined with APSFX$ to force a name to have a suffix according to the current file naming 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$ can then be used to return the filename plus the suffix required for the next step.

The following restrictions apply when using the SRSFX$ call:

- 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.
- If the suffix list contains .F77, a pathname such as pathname>.F77 is treated as a valid suffix found; that is, .F77. The filename returned is the null string ( ).
- If the filename and suffix exceed 32 characters or the pathname and suffix exceed 128 characters, a search with suffix is not done and the next suffix is attempted. For example, a filename of 32 characters is simply 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.

Program Using SRSFX$ and CL$PIX: Here is an example of a simple program that uses SRSFX$ to check on the existence of a file. It also uses the CL$PIX routine.

```
main:
        proc;

$Insert syscom>keys.ins.pl1

$Insert syscom>errd.ins.pl1

/* External entry points */

dcl srsfx$ entry (fixed bin, char(*)var, fixed bin,
                  fixed bin,fixed bin, (1) char(32)var,
                  char(32)var,fixed bin, fixed bin),
    cl$get entry (char(*)var, fixed bin, fixed bin),
    cl$pix entry (bit(16) aligned, char(*)var, ptr,
                  fixed bin, char(*)var, ptr, fixed bin,
                  fixed bin, fixed bin, ptr),
    er$print entry (fixed bin, char(*) var, fixed bin,
         char(*) var, char(*) var),
    tnoua entry (char(*), fixed bin),
    todec entry (fixed bin),
    tnou entry (char(*), fixed bin);
```

```
/* Local declarations */

dcl 1 bvs based,           /* Based Varying String */
       2 len fixed bin,
       2 chars char (128);
dcl pathname char(80)var,
    dir_name char(80)var,
    fil_name char(80)var,
    unit fixed bin,
    type fixed bin,
    num_suff fixed bin,
    suff_list (10) char(32)var,
    suff_used fixed bin,
    status fixed bin,
    code fixed bin,
    non_st_code fixed bin,
    pix_index fixed bin,
    bad_index fixed bin,
    picture char(30)var,
    pic_ptr ptr,
    out_ptr ptr,
    arg_line char(150) var;

dcl 1 args,
       2 dir char(128) var,
       2 file char(32) var;

/* PROMPT USER FOR ARGUMENTS */

call tnoua (’Enter directory pathname and filename
                  arguments:’, 49);

/* READ IN ARGS TO CALL */

call cl$get (arg_line, 150, code);
if code ^= 0
   then call er$print(k$nrtn, ssc$errd, code,
                           ‘CANNOT READ ARGS’,‘test’);

   else do;

/* SET UP DATA FOR CL$PIX */

       picture = ‘tree; entry; end’;
       pic_ptr = addr(picture);
       out_ptr = addr(args);
```

```
/* CALL CL$PIX TO PARSE ARGUMENTS */

     call cl$pix(0, 'test', pic_ptr, 30, arg_line, out_ptr,
                  pix_index, bad_index, non_st_code, null());
     if non_st_code ^= 0
        then do;
        call tnoua ('CANNOT PARSE ARGS, error code = ',32);
        call todec (non_st_code);
        call tnou(' ', 1);
        end;
     else do;

/* CHECK FOR EXISTENCE OF FILE IN SON, FATHER, GRANDFATHER ORDER */

        unit = 2;
        num_suff = 3;
        suff_list(1) = '.SON';
        suff_list(2) = '.FATHER';
        suff_list(3) = '.GRANDFATHER';

        pathname = dir || '>' || file;
        call srsfx$(k$exst, pathname, unit, type, num_suff,
                    suff_list, file, suff_used, status);
        if status > 0
           then call er$print(k$irtn, ssc$errd, status,
                     addr (pathname) -> bvs.chars, '');
        else do;
           if suff_used = 0
              then do;
             call tnoua('base file name only found: ', 27);
             call tnou(addr(pathname) -> bvs.chars,
                          addr(pathname) -> bvs.len);
              end;
           else do;
              pathname = pathname || suff_list(suff_used);
              call tnoua (addr(pathname) -> bvs.chars,
                          addr(pathname) -> bvs.len);
              call tnou (' form of file name found', 24);
              end;
           end;
        end;
     end;
  end;
```

This program gives the following output if the '.SON' form of the file exists:

```
R TEST
Enter directory pathname and filename arguments: TEST_UFD TEST_FILE
TEST_UFD>TEST_FILE.son form of file name found
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

TNCHK$

Verifies that a supplied string is a valid pathname.

Usage

```
DCL TNCHK$ ENTRY (FIXED BIN, CHAR(*)VAR)
                     RETURNS (BIT(1));
```

name_ok = TNCHK$ (***key, pathname***);

Parameters

key

INPUT. Determines the restrictions to be placed on the name. Keys can be added together. Possible values are

K$UPRC	Change name to uppercase before checking.
K$WLDC	Allow wildcard characters in name.
K$NULL	Allow a null pathname.

pathname

INPUT. Must follow the rules for pathnames given in the *PRIMOS User's Guide*, modified by the *key* above.

name_ok

RETURNED VALUE. Set to TRUE (1) if the name is valid given the restrictions of the keys; otherwise, set to FALSE (0).

Discussion

TNCHK$ verifies that a pathname conforms to the rules for constructing pathnames, as outlined in the *PRIMOS User's Guide*. TNCHK$ does not verify that an object represented by *pathname* actually exists.

Note that TNCHK$ always verifies pathnames that contain numeric components, such as the following pathname:

```
FRED>1>2
```

Entrynames within the pathname can be checked individually for validity by calls to FNCHK$, described earlier in this chapter.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

TSRC$$

TSRC$$ opens a file anywhere in the PRIMOS file structure.

This subroutine is considered obsolete, and its use in new programming is discouraged. Use SRSFX$ instead. Users maintaining existing programs that call TSRC$$ can refer to Appendix A for a complete description of the subroutine.

UNITS$

Returns the minimum and maximum file unit numbers currently in use by this user.

Usage

DCL UNITS$ ENTRY (FIXED BIN (15), FIXED BIN (15));

CALL UNITS$ (*min_unit, max_unit*);

Parameters

min_unit

OUTPUT. Lowest numbered file unit currently in use by this user.

max_unit

OUTPUT. Highest numbered file unit currently in use by this user.

Discussion

Although normal file unit numbers always start at 1, PRIMOS uses some negative unit numbers for internal purposes. Therefore, the minimum unit number can be negative (and is –5 for Revision 20.2).

The numbers returned in *min_unit* and *max_unit* do not imply that all intervening numbers are currently associated with this (or any other) user. Nor is it possible, using this call, to determine which of the intervening numbers are or are not in use by the caller.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

WILD$

Returns a logical value indicating whether a wildcard name was matched.

Usage

```
DCL WILD$ ENTRY (CHAR(32) VAR, CHAR(32) VAR, FIXED BIN)
          RETURNS (BIT(1) ALIGNED);
```

did_match = (***wildname, entryname, code***);

Parameters

wildname

INPUT. Wildcard name to match.

entryname

INPUT. Entryname against which to match.

code

OUTPUT. Standard error code.

did_match

RETURNED VALUE. Match found if returned value is 1; match not found if returned value is 0.

Discussion

Matching is done according to standard PRIMOS wildcard matching rules. For a description of wildcard names, refer to the *PRIMOS User's Guide*.

It is not necessary for *entryname* to exist. WILD$$ simply performs a textual manipulation of the two specified names.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

WTLIN$

Writes a line of characters to a file in compressed ASCII format.

Usage

DCL WTLIN$ ENTRY (FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN);

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

Parameters

funit

INPUT. File unit on which the file to be written is open for writing.

buffer

INPUT. Array of *count* halfwords from which the line of characters is to be written. It should contain two characters per halfword.

count

INPUT. The size of *buffer* in halfwords.

code

OUTPUT. Standard error code. The following codes are added at Rev. 22.0:

E$ZERO	Indicates that the system found and zeroed out an uninitialized block in a file on a robust partition at runtime.
E$IFCB	The disk does not contain enough free contiguous blocks.

Discussion

Information is written on the disk in compressed ASCII format. Multiple blank characters are replaced by the control 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.

The E$ZERO error can result from improper shutdown of a disk. When a system halt occurs while a file is being extended logically on a robust partition, not all data blocks may be written to disk. As a result, there may be uninitialized blocks in the file. FIX_DISK –FAST will not detect the uninitialized blocks because it does not check the validity of the data in the file. When the user tries to read the uninitialized blocks at runtime, PRIMOS determines that the blocks are invalid, zeros them, and returns E$ZERO. E$ZERO is returned only the first time that the block is read or part of the block is written. When an uninitialized block is detected during a write, the data in the buffer will be written, but any remaining space in the block will be zeros.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: No special action.

CAM File Subroutines

The Contiguous Access Method (CAM) is a way to organize files physically on disk. CAM can provide efficient data retrieval for large database applications.

Structure of CAM Files

A CAM file is stored on disk in groups of blocks known as **extents**. The blocks within an extent are contiguous, but the extents themselves are stored wherever contiguous space is available. Each block in an extent contains 1024 halfwords. A CAM file's physical end of file (PEOF) is at the end of the last block allocated to the file.

The maximum number of extents that a CAM file can contain depends on the master disk revision of PRIMOS. On pre–Rev. 22.1 disk partitions, a CAM file can contain a maximum of 340 extents. On Rev. 22.1 and later disk partitions, a CAM file can contain a maximum of 16,381 extents.

On robust partitions, SAM and DAM files are organized physically as CAM files. However, all files on robust partitions retain the file type attribute (SAM, DAM, or CAM) specified by the user at file creation. For this reason, applications that deal only with SAM and/or DAM files can, even on robust partitions, run without alteration and create and access SAM and DAM files. System subroutines and commands, such as SRCH$$ and LD, always report the file type as specified by the user at file creation.

The efficiency with which you can access CAM files increases as extents become fewer in number and larger in size. For this reason, it is good practice to monitor large CAM files, using the LEM command. If the LEM command indicates that the file has a large number of extents, follow these steps to reduce the number of extents:

1. Use the LCB command to find out whether there is enough free contiguous space to reduce the number of extents. If there is enough space, copy the file into a second file, delete the first file, and change the name of the second file to the original name.
2. Use MAGSAV to save the contents of the partition on tape, remake the partition, and use MAGRST to restore the contents of the partition.
3. If you do not have time to do step 2, you can use MAGSAV to save the CAM file or files in question. Then delete the CAM file(s). Use the MAGRST command to restore the CAM file(s). You can then use the LEM command to find out whether the file still has a large number of extents. If it still has a large number of extents, perform step 2.

See the *Operator's Guide to File System Maintenance* for information about the LCB command. See the *PRIMOS Commands Reference Guide (*Rev. 22.0 or later) for information about the LEM command.

CAM File Extent Maps

For every CAM file, there are one or more **extent maps** on disk that enable the system to access the CAM file. A CAM file has one extent map for each 340 extents or fraction of 340. Extent maps contain the following information:

- A header that gives the address of the logical end of the file, the number of extents in the file, and the file's **extent length value**. A CAM file's logical end of file (LEOF) is equivalent to the LEOF of SAM and DAM files. Allocation size values are discussed below. This information is found only in the header of the first extent map of a CAM file; headers in any subsequent extent maps contain zeros.
- A series of entries for individual extents in the file. Each entry indicates the starting location (beginning record address) of the first record in an extent, and the number of records in the extent.

The structure of the on–disk extent map is subject to change without notice in future revisions of PRIMOS. Any changes to the structure of the extent map data returned by CF$REM will be noted in this chapter.

Creating CAM Files

To create a file specifically as a CAM file, use the subroutine SRCH$$ with the K$NCAM key. A CAM file is created with 1 extent of 1 block. SRCH$$ also opens and closes CAM files.

Truncating CAM Files

The subroutine PRWF$$, when called with the K$TRNC key, truncates a CAM file logically, but does not move the file's PEOF. The subroutine CF$EXT can truncate a CAM file physically to a specified position. If the specified position is before the LEOF, CF$EXT sets the LEOF to the PEOF. Physically truncating a file to its current LEOF can prevent the waste of disk space that occurs when the PEOF extends beyond the logical end.

Extending CAM Files

The subroutines PRWF$$ and WTLIN$ extend a CAM file by whatever amount of space is required for the data that is to be added to the file. The user does not specify the amount of space by which the CAM file is to be extended.

CF$EXT extends a CAM file by the number of blocks that the user specifies in the call to CF$EXT.

Three user options available beginning at Rev. 22.0 affect the way in which the system extends a CAM file.

Maximum Extent Size: A partition option that specifies the maximum number of blocks to be added to a CAM file at each allocation, when the CAM file is larger than the maximum extent size. The default maximum extent size is 256 for robust partitions and 32 for nonrobust partitions. For pre–Rev. 22.0 partitions, the nonrobust partition default applies and cannot be changed by the user.

Minimum Extent Size: A partition option that specifies the minimum number of blocks to be allocated in extents added to a CAM file. The default minimum extent size is 64 for robust partitions and 16 for nonrobust partitions. For pre–Rev. 22.0 partitions, the nonrobust partition default applies and cannot be changed by the user.

Allocation Size Value: A CAM file option that specifies the amount of space added to a CAM file at each allocation. CF$SME sets the allocation size value to any number of blocks within the range 0 to 32767. A CAM file is created with an allocation size value of 0.

Note The maximum and minimum extent sizes for all CAM files on a partition are set by MAKE or FIX_DISK. See the *Operator's Guide to File System Maintenance* for information about MAKE and FIX_DISK. The maximum and minimum extent sizes cannot be set to zero. The minimum extent size can be any value that is less than or equal to the maximum extent size.

The current setting of the allocation size value determines how PRIMOS allocates additional space when a user writes beyond the physical end of the file.

- If a CAM file's allocation size value is 0, PRIMOS allocates additional space equal to the current size of the file, with the result that each allocation of new space doubles the size of the file. The system first attempts to add the space to the last extent in the file. If more space is required than can be added to the last extent, the system allocates an additional extent. If the amount of additional space requested is greater than the minimum extent size, the new extent will be at least minimum extent size. If the number of contiguous blocks available is not greater than or equal to the minimum extent size, no new extent is allocated, and the error E$IFCB (insufficient free contiguous blocks) is returned.
- The file continues to be doubled in size with each new request for space until the file reaches the maximum extent size set for the partition. When the file reaches the maximum extent size, the file is extended by maximum extent size with each request for space. Thus, on a partition with the default maximum extent size of 256 blocks, the number of blocks allocated to a CAM file by each request would be as follows: 1, 2, 4, 8, 16, ... 256, 512, 768, 1024, and so on.

- If a CAM file's allocation size value is greater than 0, PRIMOS first attempts to add space to the last extent in the file. If more space is required than can be added to the last extent, an additional extent is allocated. The total amount of space added at each allocation is equal to the allocation size value. Note that if the number of contiguous blocks available is not equal to or greater than the allocation size value, the system adds some space at the end of the last extent in the file (if any space can be added to the last extent) and returns the E$IFCB error.

This section describes subroutines that are designed to handle files that have been created specifically as CAM files. These subroutines are listed in Table 4–1.

Table 4–1. Subroutines for Files Created Specifically as CAM Files

Routine	*Function*
CF$EXT	Move a CAM file's physical end of file (PEOF) by a user–specified number of blocks. If truncating, move the file's physical end; if the new physical end is before the logical end, set logical end to physical end.
CF$REM	Return information about a CAM file's physical layout on disk.
CF$SME	Set a CAM file's allocation size value.

CF$EXT

Moves a CAM file's physical end of file (PEOF).

Usage

```
DCL CF$EXT ENTRY (FIXED BIN(15), FIXED BIN(31),
                              FIXED BIN(31), FIXED BIN(15));

CALL CF$EXT (unit, requested_peof, actual_peof, code);
```

Parameters

unit

INPUT. The unit on which the file is open.

requested_peof

INPUT. The requested new location of the PEOF. The location is expressed as an offset in halfwords (16 bits) from the beginning of the file.

actual_peof

OUTPUT. The location of the PEOF after the call to CF$EXT has been completed. The location is expressed as an offset in halfwords from the beginning of the file.

code

OUTPUT. The standard error code. Possible values are

E$OK	The call to CF$EXT was executed without error.
E$WFT	The file is not a CAM file.
E$IFCB	The disk does not contain enough free contiguous blocks.
E$UNOP	The file is not open.
E$IMEM	There is not enough memory for the extent map.
E$BPAR	There is an invalid input parameter; generally, an invalid PEOF value.
E$BKIO	The file is open for block mode; CF$EXT cannot truncate it.

E$EXMF The file cannot be extended because the extent map is full. At Rev. 22.1, an extent map can contain a maximum of 16,381 entries. On revisions earlier than 22.1, an extent map can contain a maximum of 340 entries.

Discussion

CF$EXT extends or truncates a file created specifically as a CAM file. To extend or truncate a file, CF$EXT moves its PEOF. The PEOF is the last block allocated to the file.

When CF$EXT truncates a CAM file to a location before the logical end of file (LEOF), it sets the LEOF to the physical end of the file. Note that PRWF$$ can truncate a file only logically.

Loading and Linking Information

The dynamic link for CF$EXT is in PRIMOS.

Effective for PRIMOS Revision 20.0 and subsequent revisions.

CF$REM

Returns information about a CAM file's physical layout on disk.

Usage

DCL CF$REM ENTRY (FIXED BIN(15), (*)FIXED BIN(15),
FIXED BIN(15), FIXED BIN(15)
[, FIXED BIN(15)]);

CALL CF$REM (*unit, buffer, length, code,* [*nmap*]);

Parameters

unit

INPUT. Unit on which the file is open.

buffer

INPUT. User buffer into which the extent map is to be copied.

length

INPUT. Length of the user buffer, in 16–bit halfwords. The length is stored as an unsigned integer.

code

OUTPUT. The standard error code. Possible values are

E$OK	The call to CF$REM was executed without error.
E$WFT	The file is not a CAM file.
E$UNOP	The file is not open.
E$BPAR	Invalid extent map number is specified in *nmap*. Either the number in *nmap* is negative, or no extent map with that number exists.
E$BFTS	The buffer is too small, and CF$REM did not return the extent map. Beginning at PRIMOS Rev. 22.1, CF$REM returns the extent map's header when CF$REM returns E$BFTS, provided that the user buffer is at least four halfwords long. Specify buffer size in the *length* parameter (see above).

nmap

OPTIONAL INPUT. The number of the particular extent map that CF$REM is to return. If you omit this parameter or specify 0 for it, CF$REM returns the entire contents of the file's extent map or maps. A CAM file's extent maps are numbered sequentially, beginning at 1. Thus, the extent maps of a file with three maps are numbered 1, 2, and 3.

Discussion

CF$REM returns to a user buffer information taken from the on–disk extent map of a file created specifically as a CAM file. This information describes the CAM file's physical layout on disk.

The program that calls CF$REM must declare the extent map data structure as follows:

```
DCL 1 extent_map,
      2 header,
        3 leof fixed bin(31),
        3 min_ex_len fixed bin(15),
        3 num_extents fixed bin(15),
      2 table(num_extents),
        3 ex_bra fixed bin(31),
        3 ex_len fixed bin(15);
```

The extent map data structure contains the following data items:

leof

OUTPUT. Logical end of the file (end of data storage), in 16–bit halfwords.

min_ex_len

OUTPUT. Allocation size value, in blocks.

num_extents

OUTPUT. Number of extents in the file.

ex_bra

OUTPUT. Physical block address of the first record in the extent (beginning record address).

ex_len

OUTPUT. Length of the extent, in blocks.

If the user buffer is not large enough to receive information about all of the extents of the file, CF$REM returns the error code E$BFTS, and does not return any information about extents. To correct this error, compute the amount of buffer required using the formula:

```
map_size (in halfwords) = hdr_size + (num_extents *
          table_entry_size)
```

where

```
         hdr_size = 4
table_entry_size = 3
      num_extents = the value returned by the call to
                   CF$REM
```

Then call CF$REM again with *length* set to the number of halfwords indicated by this formula.

Note the following changes to CAM files and CF$REM at Rev. 22.1:

- A CAM file can contain a maximum of 16,381 extents.
- A CAM file has one extent map for every 340 extents or fraction of 340. A file with 340 or fewer extents has only one extent map, which contains a header and information about the extents in the file. If a CAM file has more than one extent map, the first extent map contains the header, followed by information about the first 340 extents in the file. Each subsequent extent map contains a header filled with zeros followed by information about individual extents.
- CF$REM returns the extent map header when E$BFTS is reported, provided that you have allocated at least four halfwords for the user buffer. For this reason, it is good practice to specify a buffer size of at least four halfwords in your call to CF$REM.

At Rev. 22.1 and later revisions, it is good practice initially to call CF$REM to return the first or only extent map of a file. The header information in the first map enables you to determine whether the file has more than one extent map. If necessary, you can then call CF$REM again to access other extent maps by number.

Loading and Linking Information

The dynamic link for CF$REM is in PRIMOS.

Effective for PRIMOS Revision 20.0 and subsequent revisions.

CF$SME

Sets the allocation size value of a CAM file.

Usage

```
DCL CF$SME ENTRY (FIXED BIN(15), FIXED BIN(15),
                  FIXED BIN(15));
```

CALL CF$SME (*unit, ext_length_val, code*);

Parameters

unit

INPUT. The unit on which the file is open.

ext_length_val

INPUT. The new allocation size value for this file.

code

OUTPUT. The standard error code. Possible values are

E$OK	The call to CF$SME was executed without error.
E$WFT	The file is not a CAM file.
E$UNOP	The file is not open.
E$BPAR	There is an invalid input parameter.

Discussion

CF$SME sets the allocation size value of a particular file that was created specifically as a CAM (Contiguous Access Method) file. The allocation size value is the number of blocks to be allocated to a file when additional space is required. CF$SME can be called to modify the allocation size value at any time after the file has been created, provided that the file unit is open for write access. CF$SME cannot change the default minimum extent size of a partition.

The appropriate allocation size value to set for a file depends on how the file is used and the partition's default maximum and minimum extent size values. For a small file, the partition defaults may be too large. A large file may require a larger allocation size value than the partition default to ensure that enough space is allocated for the file.

For example, suppose you want to create a file that has 100,000 blocks on a partition that has the default minimum extent size (64 blocks) and maximum extent size (256 blocks). Given the maximum extent size, you may not be able to create a file that large unless the disk is relatively clean. However, by using CF$SME to set the file's allocation size value to 1000 blocks, you increase the amount of space that can be added to the file at each allocation. Increasing the file's allocation size value in this way makes it less likely that the limit on the number of extents per CAM file will prevent you from creating your file on this partition.

Loading and Linking Information

The dynamic link for CF$SME is in PRIMOS.

Effective for PRIMOS Revision 20.0 and subsequent revisions.

5 EPF Management

This chapter describes the group of subroutines that support the PRIMOS Executable Program Format (EPF) mechanism. EPFs and their operation are described in detail in the *Advanced Programmer's Guide I: BIND and EPFs*; how to create them and make them accessible for execution is described in the *Programmer's Guide to BIND and EPFs.*

EPF execution consists of allocating virtual memory space in which the EPF can run and store its data; mapping the EPF to virtual memory; initializing the EP3's linkage areas; and finally, invoking, or starting the execution of, the EPF. Subroutines are provided to perform each of these functions separately.

Also provided is a subroutine that combines, in a single call, all of the above functions, as well as subroutines used for housekeeping of EPFs and their virtual memory segments.

Rev. 23.0 and later versions of PRIMOS make registered EPFs available to the user. The registration of EPFs offer the following advantages:

- Shared linkage, thus reducing the system working set
- Pre-snapped dynamic links, reducing execution time
- Faster initialization of per–user data, reducing startup time

Registered EPFs are maintained in shared address space and are listed in a special registered EPF database. When the System Administrator registers an EPF, PRIMOS automatically allocates space for it from the available shared dynamic segments. PRIMOS also carries out many dynamic linking and initialization tasks at registration time.

Registered EPFs are especially useful for programs and libraries that are widely used on a system. As a result of the reduced initialization and linking overhead, registered EPFs can be more efficient than nonregistered EPFs for many applications. Because at least part of a registered EPF is mapped to shared memory, registered EPFs occupy less space in the user's private address space.

For a more detailed description of how registered EPFs function, see the *Advanced Programmer's Guide I: BIND and EPFs.*

The following subroutines, their declarations, and their calling sequences are described in this chapter:

EPF$ALLC	Perform the linkage allocation phase for an EPF.
EPF$CPF	Return the state of the command processing flags in an EPF.
EPF$DEL	Deactivate the most recent invocation of a specified EPF.
EPF$INIT	Perform the linkage initialization phase for an EPF.
EPF$INVK	Initiate the execution of a program EPF.
EPF$ISREADY	Indicate that a given registered EPF is ready or suspended.
EPF$MAP	Map the procedure images of an EPF file into virtual memory.
EPF$REG	Register an EPF in the appropriate registered database.
EPF$RUN	Combine functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.
EPF$UREG	Unregister an EPF by removing it from its address space.
LN$SET	Modify a user's search rules to allow dynamic linking to a library EPF.
REMEPF$	Remove an EPF from a user's address space.
RPL$	Replace one EPF runfile with another.

EPF$ALLC
EPF$AL

Performs the linkage allocation phase for an EPF.

Usage

DCL EPF$ALLC ENTRY (PTR OPTIONS (SHORT), FIXED BIN);

CALL EPF$ALLC (*epf_id, code*);

Parameters

epf_id

INPUT. The identifier of the mapped–in EPF (created by EPF$MAP).

code

OUTPUT. Standard error code. Possible values are

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$ILTD	An invalid EPF Linkage Table Directory (LTD) linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.
E$EPFT	An invalid EPF type field was detected when trying to allocate storage. Resubmit the file to BIND.

Discussion

The EPF$ALLC call allocates storage for the linkage and static data areas of an EPF. 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. EPFs are allocated static data and linkage area space in process–class storage. All storage is managed by PRIMOS.

Refer to the *Advanced Programmer's Guide I: BIND and EPFs* for a discussion of storage classes.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$CPF
EPF$CP

Returns the state of the command processing flags in an EPF.

Usage

```
DCL EPF$CPF ENTRY (PTR OPTIONS (SHORT),
                  1, 2, 3 BIT(1),
                        3 BIT(1),
                        3 BIT(1),
                        3 BIT(1),
                        3 BIT(4),
                        3, 4 BIT(1),
                           4 BIT(1),
                           4 BIT(1),
                           4 BIT(1),
                           4 BIT(1),
                        3 BIT(7),
                      2 FIXED BIN(15),
                  FIXED BIN(15));
```

CALL EPF$CPF (*epf_id, epf_info, code*);

Parameters

epf_id

INPUT. The identifier of the mapped–in EPF.

epf_info

OUTPUT. The structure that is to contain the EPF command processing features. The structure is described below.

code

OUTPUT. Standard error code. Possible value is

E$BPAR	An undefined value of *epf_id* was passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF$MAP.

Discussion

The command processing features that the EPF can invoke are set during the execution of the EPF linker, BIND.

The structure in which the invokeable features are returned is shown below. Refer to the *Advanced Programmer's Guide III: Command Environment* for explanations of each bit.

```
DCL 1 epf_info based,
          2 command_flags,
            3 wildcards bit(1),
            3 treewalks bit(1),
            3 iteration bit(1),
            3 verify bit(1),
            3 reserved bit(4),
            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);
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$DEL
EPF$DL

Deactivates the most recent invocation of a specified EPF.

Usage

DCL EPF$DEL ENTRY (PTR OPTIONS (SHORT), FIXED BIN(15));

CALL EPF$DEL (*epf_id, code*);

Parameters

epf_id

INPUT. The identifying number of the EPF to be deactivated. This number is supplied by the EPF$MAP subroutine (described later in this chapter).

code

OUTPUT. Standard error code. Possible values are

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.
E$SWPR	An attempt was made to delete an EPF that is suspended in the calling process.

Discussion

The EPF$DEL subroutine deactivates one invocation of an EPF for the calling process. The segment(s) used for linkage and static data for the most recent invocation of the EPF are returned to the free pool of dynamic segments. If this EPF has not been previously executed by a call to EPF$INVK, the EPF procedure segment(s) are released, and the storage used by the in–memory EPF database is released.

The invocation of an EPF uses 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 again in a relatively short time.

If the EPF invocation is not terminated by a call to EPF$DEL, system segments are not returned to the free segment pool, and a user may eventually run out of segments in the dynamic segment range.

If an error occurs while attempting to return EPF procedure segments to the system, the message

```
Unable to free EPF procedure segments
```

is displayed, and the user's command environment is reinitialized.

Any error detected while deallocating storage causes an appropriate error message to be displayed at the user's terminal and the user's command environment to be reinitialized.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$INIT
EPF$NT

Performs the linkage initialization phase for an EPF.

Usage

```
DCL EPF$INIT ENTRY (FIXED BIN(15),PTR OPTIONS (SHORT),
                      FIXED BIN(15));

CALL EPF$INIT (key, epf_id, code);
```

Parameters

key

INPUT. Specifies the action to be performed. Possible values are

K$INITALL (K$INAL for FTN callers) — Specifies complete initialization of data areas.

K$REINIT (K$REIN for FTN callers) — Specifies reinitialization of only the data areas. EPF$INIT reinitializes only the static data and faulted indirect pointers (IPs), but maintains other data such as resolved IPs and entry control blocks.

epf_id

INPUT. The identifier of the mapped–in EPF (supplied by EPF$MAP, described later in this chapter).

code

OUTPUT. Standard error code. Possible values are

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, probably an attempt to reinitialize before a complete initialization was done.

E$ILTE — An invalid EPF Linkage Table Entry (LTE) linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.

E$ILTD — 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.

Discussion

The EPF must already be mapped to memory (by EPF$MAP), with its static data areas already allocated (by EPF$ALLC).

The EPF$INIT call must be made with a *key* value of K$INITALL before any call is made with a *key* value of K$REINIT; that is, a complete initialization of a mapped and allocated EPF must have been performed at least once before a reinitialization can be done.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$INVK
EPF$VK

Initiates the execution of a program EPF.

Usage

```
DCL EPF$INVK ENTRY (PTR OPTIONS (SHORT), FIXED BIN(15));

CALL EPF$INVK (epf_id, code);
  (or)
DCL EPF$INVK ENTRY (PTR OPTIONS (SHORT), FIXED BIN(15),
                    CHAR(1024) VAR, FIXED BIN(15),
                    1, 2 CHAR(32) VAR,
                      2 FIXED BIN(15),
                      2 PTR OPTIONS (SHORT),
                      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),
                        3 FIXED BIN(31),
                        3 FIXED BIN(31),
                        3 FIXED BIN(31),
                        3 FIXED BIN(31),
                    1, 2 BIT(1),
                      2 BIT(1),
                      2 BIT(14),
                    PTR);

CALL EPF$INVK (epf_id, code, com_args, ret_code, com_state, flags,
                    rtn_function_ptr);
```

Parameters

epf_id

INPUT. The identifier of the EPF (supplied by EPF$MAP, described later in this chapter).

code

OUTPUT. Standard error code. Possible values 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. Arguments to the invoked EPF.

ret_code

OUTPUT. Return code from execution of the invoked EPF. Any standard error code generated during program execution may be returned. Refer to the *Advanced Programmer's Guide III: Command Environment* for a complete list.

com_state

INPUT. Contains information relative to the EPF invocation. The format is described in the Discussion section.

flags

INPUT. Contains information relative to the command function invocation. The format is described in the Discussion section.

rtn_function_ptr

OUTPUT. Pointer to a return function structure used by an EPF acting as a function. The format is described in the Discussion section.

Discussion

Program EPFs written as programs (that is, expecting no command arguments and returning no error code) are normally invoked with the first calling sequence shown in the Usage section above. Program EPFs written as functions, and those expecting arguments, must be invoked using the second calling sequence.

The additional arguments are provided for passing invocation information to the program being invoked, and for returning data to the invoking program.

The *Advanced Programmer's Guide I: BIND and EPFs* contains a full description of the ways in which EPFs can be invoked from within other programs.

Before the EPF$INVK call is made, the EPF must have been mapped into virtual memory and the static data areas must be both allocated and initialized. The required order of calls is EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK.

The address of the starting Entry Control Block (ECB) 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.

The calling program supplies in *com_state* information required by the invoked EPF when it expects arguments or when it is called as a function. The format of *com_state* is shown below.

```
DCL 1 com_state,
      2 com_name char(32) var,
      2 version fixed bin(15),
      2 vcb_ptr ptr,
      2 reserved_1 fixed bin(15),
      2 cp_iter_info,
        3 mod_after_date fixed bin(31),
        3 mod_before_date fixed bin(31),
        3 bk_after_date fixed bin(31),
        3 bk_before_date fixed bin(31),
        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 reserved_2 bit(11),
        3 verify_sw bit(1),
        3 botup_sw bit(1),
        3 reserved_3 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 reserved_4 bit(13),
        3 created_after_date fixed bin(31),
        3 created_before_date fixed bin(31),
        3 accessed_after_date fixed bin(31),
        3 accessed_before_date fixed bin(31);
```

The level–2 fields in the previous calling program have the following meanings:

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 Pointer 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 invoked EPF from the calling program. The last four date fields are valid only at Rev. 20.0 and later.

The *flags* argument informs the called EPF that it is being called as a function, and that it is expected to return a function value; it has the following 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 remaining fifteen bits are undefined. The format of the structure pointed to by the *rtn_fcn_struc* pointer is

```
1 rtn_fcn_struc,
    2 version  fixed bin (15),
    2 value_str  char (*) var;
```

The version must be set to zero by the called EPF. The memory space for this data will have been allocated by the EPF. The caller uses this data and later deallocates the memory space using FRE$RA.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$ISREADY

Returns information on whether a registered EPF is ready or suspended.

Usage

```
DCL EPF$ISREADY ENTRY (CHAR(32) VAR, FIXED BIN(15),
            FIXED BIN(15), BIT(1) ALIGNED, FIXED BIN(15));

CALL EPF$ISREADY (epfname, epf_database, epf_level, ready_flag,
                  status);
```

Parameters

epfname

INPUT. The name of the registered EPF to be checked.

epf_database

INPUT. The database to be checked. Currently, only K$PUBLIC is a valid database.

epf_level

INPUT. A number indicating how far down the EPF Information Table (EIT) chain to search. This allows the user to check the status of any version of the registered EPF. There is no limit on the number of versions of the same EPF that can be registered at the same time.

ready_flag

OUTPUT. This flag is set to true if the EPF is ready.

status

OUTPUT. Standard error code.

Discussion

The EPF$ISREADY subroutine retrieves a name block pointer for the specified EPF and follows the EPF Information Table (EIT) chain until the proper EPF is located. It then sets *return_flag* dependent on the state of the EPF.

Loading and Linking Information

V–mode and I–mode: No special action.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

EPF$MAP
EPF$MP

Maps the procedure images of an EPF file into virtual memory.

Usage

```
DCL EPF$MAP ENTRY (FIXED BIN(15), FIXED BIN(15),
                   FIXED BIN(15), FIXED BIN(15)) RETURNS
             (PTR OPTIONS (SHORT));
```

epf_id = EPF$MAP (*key, unit, access_rights, code*);

Parameters

key

INPUT. Segment mapping options. Possible values are

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.

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$R gives only read access; it does not permit execution. K$RX give execution access and also implies read access. Use K$RX to be assured of future compatibility.

code

OUTPUT. Standard error code. See the Discussion section.

epf_id

RETURNED VALUE. The identifier of the mapped–in EPF. This identifies the in–memory EPF when calling other EPF$ subroutines. If an error is returned to the caller, *epf_id* is undefined.

Discussion

The EPF$MAP subroutine is called to perform the map–to–memory function of the EPF mechanism. The EPF file must already have been opened for VMFA–read on a file unit; that is, you must first call either SRCH$$ or SRSFX$ with the K$VMR key specified. Refer to Chapter 4 for descriptions of these subroutines.

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 the *Advanced Programmer's Guide III: Command Environment* for more information on program and library EPFs.

The EPF must be mapped to memory in order to be executed. The user code that calls EPF$MAP or EPF$RUN (described later in this chapter) should be capable of dealing with any error condition that might result when the EPF is invoked.

If an error occurs while attempting to allocate dynamic memory space for the EPF or if the user's command environment becomes corrupted, an 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 (for example, a circular list is detected), 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:

Error Code	*Meaning*
E$NMVS	Insufficient VMFA segments available for EPF mapping. Caller must either wait until some VMFA segments are returned to the free pool, (by this user or by others), or request that the system be reconfigured to allow the caller more VMFA segments.
E$NMTS	Insufficient user segments for copying EPF to memory from a remote node or using the K$COPY key.
E$ROOM	Insufficient dynamic storage is available.

Note In response to any of these three messages, the user can release temporary segments by

- Reentering a suspended subsystem via the REENTER command
- Deactivating previous EPF invocations via the REMEPF command
- Releasing command levels via the RELEASE_LEVEL command
- Reinitializing the command environment via the ICE command (as a last resort)

Error Code	*Meaning (continued)*
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 cannot 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.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$REG

Enables registration of EPFs by opening the specified file, mapping it to memory, allocating linkage segments, initializing the linkage segments, and registering the EPF in the appropriate registered database.

Usage

DCL EPF$REG ENTRY (PTR, FIXED BIN(15));

CALL EPF$REF (*register_info_ptr, status*);

Parameters

register_info_ptr

INPUT. A structure describing the EPF to be registered.

status

OUTPUT. Standard error code.

Discussion

The structure of the returned *register_info* entry is

```
DCL 1 register_info,
          2 pathname CHAR(128) VAR,
          2 init_comline CHAR(1024) VAR,
          2 epf_database FIXED BIN(15),
          2 dependency_list_ptr CHAR(32) VAR,
          2 priority_search_list_ptr CHAR(32) VAR,
          2 idl CHAR(32) VAR;
```

pathname

INPUT. The pathname of the EPF to be registered.

init_comline

INPUT. The command line for the initialization routine.

epf_database

INPUT. The database in which the EPF is to be registered. Possible value is

K$PUBLIC	Registered nonsystem class database for ring 3 EPFs.

dependency_list_ptr

INPUT. A pointer to an array of EPF names on which the specified EPF is dependent. These EPFs must be registered when the specified EPF is registered so that it is ready to be executed.

priority_search_list_ptr

INPUT. A pointer to an array of EPF names. During dynamic linking, these EPFs are searched before the libraries described within the search list.

idl

INPUT. A pointer to an array of EPF names. This array defines the order in which to invoke initial entry ECBs. This can be used to force the order of initialization.

EPF entries in the *epf_dependency_list* are placed in the Reference Linkage Table (RLT). If any of these EPFs does not exist, no attempt is made to snap the links for the EPF being registered.

If the *priority_search_list* is not null, the search rules are updated. This forces the linking mechanism to search the given EPFs within the *priority_search_list* before scanning the defined search rules. At the end of the initialization, the search rules are reset.

EPF$REG must scan the *epf_database* to complete initialization of suspended EPFs which may become ready upon registration of the current EPF. This subroutine makes calls to other subroutines to resolve as many references as possible for currently suspended EPFs.

Loading and Linking Information

V–mode and I–mode: No special action.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

EPF$RUN
EPF$RN

Combines functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.

Usage

```
DCL EPF$RUN ENTRY (FIXED BIN (15), FIXED BIN (15),
                   FIXED BIN (15))
                   RETURNS (PTR OPTIONS (SHORT));
epf_id = EPF$RUN (key, unit, code);

(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(1),
                2 BIT(14),
             PTR)
             RETURNS (PTR OPTIONS (SHORT));

epf_id = EPF$RUN (key, unit, code, com_args, ret_code, com_state,flags,
                  rtn_function_ptr);
```

Parameters

key

INPUT. Specifies action to be performed. Possible values are

K$INVK	Map, create, allocate, and initialize static data areas, and leave EPF in cache upon completion.
K$INVK_DEL (K$IVD for FTN callers)	Map, allocate, and initialize static data areas, invoke but do not cache EPF after completion.
K$REST	Map, allocate, and initialize static data areas, but do not invoke the EPF.

unit

INPUT. File unit on which the EPF is open for VMFA–read.

code

OUTPUT. Standard error code. Possible values include all error codes returned by EPFMAP, EPFALLC, EPF$INIT, or EPF$DEL.

com_args

INPUT. The command arguments.

ret_code

OUTPUT. Error code returned by invoked EPF.

com_state

INPUT. Contains information relative to the EPF invocation. See the EPF$INVK subroutine, described earlier in this chapter.

flags

INPUT. This field contains information relative to the command function invocation. See the EPF$INVK subroutine.

rtn_function_ptr

OUTPUT. Pointer to a return structure used by the EPF when called as a function. See the EPF$INVK subroutine.

epf_id

RETURNED VALUE. The identifier for the EPF created by a call to EPF$MAP from the EPF$RUN subroutine. If the EPF is deleted after its invocation completes, the *epf_id* is undefined.

Discussion

This subroutine 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$$ or SRSFX$ 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 sequence shown above. EPFs written as functions, and those expecting arguments, must be invoked using the second calling sequence. The additional arguments are provided for passing invocation information to the program being invoked, and for returning data to the invoking program.

Refer to the *Advanced Programmer's Guide III: Command Environment* for a full discussion of how to call EPFs from within other programs.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

EPF$UREG

Enables unregistration of an EPF by removing it from its address space. This subroutine decrements the user count and unregisters the EPF if the caller is the last to use an old version of the EPF.

Usage

```
DCL EPF$UREG ENTRY (FIXED BIN(15), CHAR(32) VAR,
                   POINTER OPTIONS (SHORT), FIXED BIN(15),
                   FIXED BIN(15), FIXED BIN(15));

CALL EPF$UREG (remove_key, epf_pathname, epf_smt, first_proc_seg,
               epf_database, status);
```

Parameters

remove_key

INPUT. Specifies the action to be performed. Possible values are

K$DUCT	Decrements the user–count on this EPF. This key is used only when the local Segment Mapping Table (SMT) of a registered EPF is being removed and the UNREGISTER_ EPF command is not used.
K$UNREG	Removes the entry from the EPF namespace. With this key, the associated segments for procedure and per–user linkage areas are released.

epf_pathname

INPUT. The pathname of the EPF to be unregistered. If *epf_pathname* is a publicly registered EPF, *epf_pathname* is a file entry only.

epf_smt

INPUT. The local SMT pointer, which is passed by REMEPF_ if the REMEPF command is used. If the call is made from a user's program, this pointer must be either null or a valid SMT pointer.

first_proc_seg

INPUT. The first procedure segment number of the EPF. Each different version of an EPF can be uniquely identified by its first procedure segment. Use the LIST_REGISTERED_EPF command to display the first procedure segment number of an EPF.

epf_database

INPUT. The database in which the EPF was registered. Possible value is

K$PUBLIC	Registered nonsystem class database for ring 3 EPFs.

status

OUTPUT. The status of the remove operation.

Discussion

This subroutine removes the specified registered EPF from the registered EPF database without having to unregister any other EPFs that may be linked to the EPF in question.

Loading and Linking Information

V–mode and I–mode: No special action required.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

LN$SET

Modifies a user's search rules and other data structures to allow dynamic linking to a library EPF.

Usage

```
DCL LN$SET ENTRY (POINTER OPTIONS (SHORT),
                  FIXED BIN(15));

CALL LN$SET (smtp, code);
```

Parameters

smtp

INPUT. A pointer to a segment mapping table for a given EPF library.

code

OUTPUT. The status code. Possible values are

E$OK	The call to LN$SET was completed successfully.
E$BPAR	Null SMTP.
E$BVER	EPF is not a library
E$BDAT	There are no entries in the library.
E$NDAT	The EPF has no library information.
E$NINF	No search rule was found for this library.

Discussion

LN$SET modifies a user's search rules and other data structures so that a library EPF that has been mapped–in by the Source Level Debugger (DBG) can be linked dynamically.

LN$SET tries to place into its caller's entrypoint search list the identifier for the library to be debugged. This identifier is the segment mapping table pointer (SMTP). LN$SET first verifies that the passed SMTP is valid and points to a mapped–in library. Then it searches the user's current entrypoint search list for an entry corresponding to the name of this library. If an entry is found, then the SMTP is inserted into the list. If not, then an error is returned.

The SMTP is obtained when DBG (or some other program) calls EPF$MAP to bring the library into memory. The SMTP is returned by EPF$MAP.

Loading and Linking Information

The dynamic link for LN$SET is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

REMEPF$

Removes an EPF from a user's address space.

Usage

```
DCL REMEPF$ ENTRY (FIXED BIN(15), CHAR(*) VAR,
                   FIXED BIN(15));

CALL REMEPF$ (key, epf_treename, code);
```

Parameters

key

INPUT. Force–delete indicator. Possible values are

K$FRC_DEL (K$FRDL for FTN callers)	Forcibly remove if process–class library EPF is initialized.
K$NO_FRC_DEL (K$DL for FTN callers)	Do not forcibly remove if process–class library EPF is initialized.

epf_treename

INPUT. Pathname of the EPF to be removed.

code

OUTPUT. Standard error code. Possible values are

E$BPAR	Invalid key specified.
E$NTA	EPF not active for this user.
E$SWPR	EPF suspended in this user's process.
E$ILTE	Invalid EPF LTE linkage descriptor.
E$ILTD	Invalid EPF LTD linkage descriptor.

Discussion

The REMEPF$ call removes either a program EPF or a library EPF from the user's address space. If the EPF is a process–class library EPF, all existing links to it from other process–class library EPFs are unsnapped.

The EPF to be removed must have a name that ends in either the .RUN or the .RP*n* suffix, where *n* is a decimal digit. Refer to the RPL$ subroutine, later in this chapter, for a discussion of the use of the RP*n* convention.

Several error conditions internal to EPF handling may result in the display to the user's terminal of error messages other than the standard PRIMOS messages given above. These errors are all considered fatal to any further processing, and result in reinitialization of the user's command environment.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

RPL$

Replaces one EPF runfile with another.

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);
```

Parameters

source_path

INPUT. Pathname of the file containing the code to be used in the new .RUN file.

target_path

INPUT. Pathname of the new .RUN file

rpl_path

OUTPUT. Pathname of the old .RUN file, which is now a .RP*n* file if it is currently in use; otherwise, a null string.

no_query

INPUT. If this bit is set, no query for changing the filename will prompt the user, and no messages are displayed. If it is unspecified by the user, the routine defaults to query displays.

code

OUTPUT. Standard error code. Possible values are

–1	Returned as a warning if at least one RP*n* file exists and is not in use.

Other standard error codes may be returned from subroutines called internally by RPL$. Refer to the *Advanced Programmer's Guide III: Command Environment* for explanations of these codes if they should be returned.

Discussion

The RPL$ 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 .RP*n*, where *n* is a decimal integer from 0 through 9. RPL$ replaces the old EPF file with this new .RUN file, but the .RP*n* file continues to exist. Users who try to access the new EPF file are linked to the new .RUN file; they may later delete or save the old version.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

6 Command Environment

User programs written in any language can make extensive use of the facilities provided by the PRIMOS command processor, including the ability to call other programs from within executing programs, to set and retrieve local and global variables, and to retrieve some of the characteristics of the user's command environment.

This chapter describes the group of subroutines that support user programs in their interaction with the PRIMOS command environment. Additional information on programming for the use of the command processor facilities can be found in the *Advanced Programmer's Guide III: Command Environment.*

The following subroutines, their declarations, and their calling sequences are described in this chapter:

CE$BRD	Return caller's maximum command environment breadth.
CE$DPT	Return caller's maximum command environment depth.
CL$PIX	Parse command arguments according to a character string "picture" of the command line.
CP$	Invoke a command from a running program.
GV$GET	Retrieve the value of a global variable.
GV$SET	Set the value of a global variable.
LIST$CMD	Return a list of commands valid at mini–command level.
LV$GET	Retrieve the value of a CPL local variable.
LV$SET	Set the value of a CPL local variable.
RD$CE_DP	Return breadth of caller's current command environment.

CE$BRD

Returns caller's maximum command environment breadth.

Usage

DCL CE$BRD ENTRY ( ) RETURNS (FIXED BIN(15));

max_ce_brdth = CE$BRD( );

Parameters

max_ce_brdth

RETURNED VALUE. Maximum number of simultaneous program EPF invocations permitted per command level.

Discussion

The CE$BRD subroutine 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 calling user. The command environment breadth is set on a per–user basis by the System Administrator.

The value returned is the same as that displayed when the LIST_LIMITS command is invoked from PRIMOS command level.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CE$DPT

Returns caller's maximum command environment depth.

Usage

```
DCL CE$DPT ENTRY ( ) RETURNS (FIXED BIN(15));

max_ce_dpth = CE$DPT( );
```

Parameters

max_ce_dpth

RETURNED VALUE. Maximum number of command levels permitted.

Discussion

The CE$DPT subroutine 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 levels permitted; that is, the command environment depth allocated to the user. The command environment depth is set on a per–user basis by the System Administrator.

The value returned is the same as that displayed when the LIST_LIMITS command is invoked from command level.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CL$PIX

Parses command arguments according to a character string picture of the command line.

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, non_std_code,
***local_vars_ptr*);**

Parameters

keys

INPUT. A 16–bit structure containing bits set to control certain details of processing. The structure can be defined in any language as a 16–bit integer whose value is determined by setting the desired bits on. (See How to Set Bits in Arguments in Chapter 1.)

The PL/I data description for this structure is

```
DCL 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 *debug* is '1'b, CL$PIX displays on the terminal a dump of the parsed argument picture. This is of limited use for most applications programs.

If *keep_quotes* is '1'b, CL$PIX does not strip quotes from parsed string arguments; otherwise, it removes one layer of quotes. This flag is ignored in CPL mode, and quotes are never stripped.

If *cpl_flag* is '1'b, CL$PIX operates in CPL mode; otherwise, it operates in normal mode. These modes are explained in detail in Appendix C.

If *pl1_flag* is '1'b, the presence of control arguments in the output structure is indicated by the PL/I data type BIT(1) ALIGNED. If *pl1_flag* is '0'b, the FORTRAN data type LOGICAL is used.

If *no_print* is '1'b, no error messages are printed by CL$PIX; only error code information is returned. If *no_print* is '0'b, *caller_name* is used to format the error message.

caller_name

INPUT. Name of the calling routine, which formats error messages if *no_print* is '0'b.

picture_ptr

INPUT. Pointer to a varying character string containing the command argument picture. If dimensioned, the array must be connected (contiguous). The syntax and semantics of the picture are defined in Appendix C.

pixel_size

INPUT. Maximum length in characters of the element(s) of the object pointed to by *picture_ptr*. This provision allows an arbitrarily large array of strings to be passed and circumvents compiler restrictions on character–string length.

com_args

INPUT. String containing the command arguments to be parsed. 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 (PCL), including the "/*" comment delimiter, are supported.

struc_ptr

INPUT –> OUTPUT. 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 in Appendix C.

pix_index

OUTPUT. Valid only when *non_std_code* is nonzero. 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.

bad_index

OUTPUT. Character index (counting from 1) of the first character of the token (word or expression) causing the error. The value of *pix_index* must be consulted to determine whether *bad_index* is relative to the command line arguments or to a pixel of the picture. *bad_index* is valid only if *non_std_code* is nonzero.

non_std_code

OUTPUT. Return code (independent of PRIMOS standard error codes), which can take on the following values:

0 No error.

1 Null argument group (two successive semicolons) in picture.

2 Missing or invalid delimiter in picture.

3 Invalid option argument name in picture.

4 Invalid repeat count in picture.

5 Unknown data type name in picture.

6 Implementation error in picture parse.

7 Token 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 cannot be given for this data type.

16 Too many instances of an option in command line.

17 Default value expression contains an undefined variable reference or a format error (CPL mode only).

18 Data typc UNCL has been given more than once or has been given for an option argument parameter.

19 Option argument begins with a dot (.), indicating a global variable, which is not allowed.

20 Undefined option argument.

21 Picture contains a numeric option argument. Option arguments must contain at least one alphabetic character.

22 Picture contains two sets of option argument names, separated by a space. Either they are instances of the same option argument and should be separated by a comma, or they are different option arguments and should be separated by a semicolon.

local_vars_ptr

INPUT/OUTPUT. Pointer used only in CPL mode. In this case, it points to the Local Variable Control Block that identifies the 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 in Appendix C.

Discussion

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.

A complete description of CL$PIX parsing syntax and rules is given in Appendix C.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

CP$

Invokes a command from a running program.

Usage

```
DCL CP$ ENTRY (CHAR(160) VAR, FIXED BIN(15), FIXED BIN(15),
              1,
                2 BIT(1),
                2 BIT(1),
                2 BIT(14),
              PTR, PTR);
```

CALL CP$ (*command_line, status, code, command_flags,*
local_variable_ptr, rtn_function_ptr);

Parameters

command_line

INPUT. Name of the command or program being invoked.

status

OUTPUT. Standard error code from CP$ subroutine execution.

code

OUTPUT. Standard error code from invoked program execution.

command_flags

INPUT. Information relative to invocation as a command function. 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 14 bits are undefined.

local_variable_ptr

INPUT. Pointer to local variables allocated during execution, if this CP$ call is made by a program executed from within a CPL file.

rtn_function_ptr

INPUT –> 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 discussion of this and other parts of the interface structure in the description of the ALC$RA subroutine in the *Subroutines Reference III: Operating System.*

Discussion

The CP$ subroutine 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.

For a detailed discussion of the use of CP$ within an EPF–based environment, refer to the *Advanced Programmer's Guide III: Command Environment.*

CP$ provides an easy–to–use interface for calling external programs. All a programmer has to do is call CP$ with an argument that represents a command line. This command line is a character string representation of the external program to be called. CP$ performs all wildcard, treewalk, and iteration processing specified by the character string; it does not, however, 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$. If the purchasing program prompts the user to insert a command line, the user can enter something like "ORDER wrench" (or the longer form shown 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 the system's command directory CMDNC0, the RESUME         */
/* command must be added to execute ORDER, which could             */
/* be one of several programs within a subdirectory                */
/* called PROGS:"RESUME PROGS>ORDER wrench."                       */

/* The subroutine cl$get is called to gather the terminal input. */

CALL CL$GET(COMMAND_LINE, COMMAND_LINE_LENGTH, CODE);

/* Now CP$ uses that command_line to fetch                         */
/* the program that will honor this request.                       */

CALL CP$(COMMAND_LINE, STATUS, CODE);
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

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*);

Parameters

var_name

INPUT. Name of the global variable whose value is to be retrieved.

var_value

OUTPUT. Returned value of variable *var_name*.

value_size

INPUT. The length of the user's buffer *var_value* in characters.

code

OUTPUT. Standard error code. Possible values are

E$BFTS	The user buffer *var_value* is too small to hold the current value of the variable. The value of the variable can be up to 1024 characters long, or, if numeric, can be between –2**31+1 and 2**31–1, inclusive.
E$UNOP	The global variable storage file is not open or is in invalid format.
E$FNTF	The variable is not found.
E$BNAM	The variable name must be preceded by a period.

Discussion

The PRIMOS command DEFINE_GVAR must be used to define the global variable file before this subroutine is called.

The name supplied in *var_name* must follow the rules for CPL global variable names and must be in uppercase. It must exist in the global variable file last invoked with DEFINE_GVAR.

Refer to the *CPL User's Guide* or the *PRIMOS User's Guide* for information on global variable usage and naming rules.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

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*);

Parameters

var_name

INPUT. Name of the global variable to be set.

var_value

INPUT. New value of the variable *var_name*.

code

OUTPUT. Standard error code. Possible values are

E$BFTS	The specified value is too big. The value of the variable can be up to 1024 characters long, or, if numeric, can be an integer between –2**31 and 2**31 – 1, inclusive.
E$UNOP	The global variable file is invalid or not open.
E$ROOM	An attempt by the variable management routines to acquire more storage fails.
E$BNAM	The variable name must be preceded by a period.

Discussion

The PRIMOS command DEFINE_GVAR must be used to define the global variable file before this subroutine is called.

The name supplied in *var_name* must follow the rules for CPL global variable names and must be in uppercase. The variable name and its new value are placed in the global variable file last invoked with DEFINE_GVAR. If the name already exists in the file, its value is overlaid by the new value.

Refer to the *CPL User's Guide* or the *PRIMOS User's Guide* for information on global variable usage and naming rules.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

LIST$CMD

Returns a list of commands valid at mini–command level.

Usage

```
DCL LIST$CMD ENTRY (CHAR(32) VAR,
                    1,
                      2 BIN(15),
                      2 BIN(15),
                      2 BIT(1) ALIGNED,
                    FIXED BIN(15));
```

CALL LIST$CMD (*wildcard_match, print_opts, code*);

Parameters

wildcard_match

INPUT. Wildcard character string that determines the subset of commands to be included in the list. Any matches found are returned herein.

print_opts

INPUT. Options to control list format, specified in the structure described in the Discussion section.

code

OUTPUT. Standard error code.

Discussion

The LIST$CMD subroutine 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 *PRIMOS User's Guide* and the *Programmer's Guide to BIND and EPFs.*

wildcard_match is a character string that is used as a pattern match for mini–level commands to be listed. The character string can contain wildcard characters. If you do not specify *wildcard_match,* LIST$CMD displays the names of all the PRIMOS commands that you can use at mini–command level.

The format in which the mini–level commands are displayed is controlled by *print_opts.* The number of lines per screen, the number of characters per line, and the presence or absence of a full–screen prompt are specified in the structure shown below.

```
DCL 1 print_opts,
      2 ll bin(15),    /* max. line length (characters) */
      2 pl bin(15),    /* max. page length (lines)    */
      2 nw bit(1) aligned, /* '1'b if no "More--" prompt */
```

The value of *ll* determines how many commands can be shown on each line of the display. The default value is 80 characters. The value of *pl* must be at least 4 in order to display a header line and at least one line of commands on one screenful. The default value is 24 lines. The standard PRIMOS `More–` prompt, which accepts the usual YES, NO, QUIT, or Return, is displayed if the value of *nw* is given as '0'b.

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.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

LV$GET

Retrieves the value of a CPL local variable.

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*);

Parameters

vcb_ptr

INPUT. Pointer to the block of local variables for the CPL program.

var_name

INPUT. Name of the variable in the CPL program.

var_value

OUTPUT. Value of the CPL variable.

var_size

INPUT. Maximum length in characters of the user buffer *var_value.*

code

OUTPUT. Standard error code.

Discussion

The LV$GET subroutine is used by CPL programs to retrieve the value of a local variable when the [GET_VAR] command function is invoked. It can also be used by user programs called from within CPL programs to perform the same function.

The caller supplies in *vcb_ptr* a pointer to the first (or only) variable control block (VCB), which is formatted as described for the LV$SET subroutine, later in this chapter.

The name supplied in *var_name* must follow the rules for CPL local variable names and must be in uppercase.

The current value of the local variable is returned to the calling program in *var_value.* The number of characters returned is either the actual number of characters in the value or the number specified in *var_size*, whichever is smaller. If the number of characters in the value is greater than that specified in *var_size*, the first *var_size* characters of the value are returned. In this case *code* indicates that the buffer size is too small.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

LV$SET

Sets the value of a CPL local variable.

Usage

DCL LV$SET ENTRY (PTR, CHAR(32) VAR, CHAR(*) VAR,
FIXED BIN(15));

CALL LV$SET (*vcb_ptr, var_name, var_value, code*);

Parameters

vcb_ptr
INPUT. Pointer to the local variable block for the CPL program.

var_name
INPUT. Name of the local variable in the CPL program.

var_value
INPUT. Value to be assigned to the CPL local variable.

code
OUTPUT. Standard error code.

Discussion

The LV$SET subroutine is used by CPL programs to set the value of a local variable when the [SET_VAR] command function is invoked. It can also be used by user programs called from within CPL programs to perform the same function.

The caller passes to LV$SET in *vcb_ptr* a pointer to the first variable control block (VCB), which has the format shown below.

```
dcl 1 vcb based,   /* Variable Manager Control Block */
  2 next_vcb ptr,  /* forward link in list of vcb's  */
  2 this_area ptr, /* ptr to area with this vcb      */
  2 var_chain ptr; /* start of var list
                      (only in 1st vcb)              */
```

Each variable in the variable storage area is represented by the structure shown below.

```
dcl 1 vh based,        /* Variable Header */
    2 next ptr,        /* forward link in list */
    2 value ptr,       /* ptr to char(n) var value */
    2 value_area ptr, /* ptr to value allocation area */
    2 value_size fixed bin, /* capacity of value in
                               chars */
    2 reserved(3) fixed bin,
    2 name char(32) var; /* name of variable being set */
```

The structures shown above are created and maintained by the variable manager when variables are defined. If the variable manager runs out of space in the current variable storage area, it attempts to allocate more space; if the attempt is unsuccessful, an error code is returned, indicating that there is no more room available.

The name supplied in *var_name* must follow the rules for CPL local variable names and must be in uppercase. The value supplied in *var_value* can be up to 1024 characters long.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

RD$CE_DP
RD$CED

Returns the breadth of the caller's current command environment.

Usage

DCL RD$CE_DP ENTRY (FIXED BIN);

CALL RD$CE_DP (*com_env_brdth*);

Parameters

com_env_brdth
OUTPUT. The current breadth of the command environment.

Discussion

The RD$CE_DP subroutine is one of several that retrieve EPF–related information from the in–memory copy of the current user's profile.

This subroutine returns the breadth of the command environment at which the user is currently operating. The breadth of the command environment is the number of program invocations at the current command level.

The maximum command environment breadth is set on a per–user basis by the System Administrator. The user can retrieve this maximum for comparison with the current breadth by using the CE$BRD subroutine, described earlier in this chapter, or by invoking the LIST_LIMITS command from PRIMOS command level.

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

7 Search Rules

The PRIMOS search rules facility enables you to set sequential search lists that PRIMOS uses to locate file system objects. The search rules facility is described in the *Advanced Programmer's Guide II: File System*. The subroutines described here permit you to read and modify these search lists, and to use search lists to locate and open file system objects.

Note The Rev. 23.0 file system introduces the concept of the **common file system name space.** The contents of the disk partitions with the common file system namespace make up **one logical entity**. This means, among other things, that logically there are no remote disks within the namespace. Be aware that with the advent of Rev. 23.0, the potential number of disks that you can access increases from 238 to 1280. Therefore, a well–considered use of the search rules subroutines may be necessary to gain more rapid access to those disks that you use most frequently.

For a detailed description of the Rev. 23.0 file system, refer to the *Advanced Programmer's Guide II: File System.*

Most search rule subroutines can be invoked by either their full name or a six–character synonym. The following subroutines, their declarations, and their calling sequences are described in this chapter.

OPSR$ Locate a file using a search list and opens the file. Create a file if the file sought does not exist.

OPSRS$ Locate a file using a search list and a list of suffixes. Open the located file, or creates a file if the file sought does not exist.

SR$ABSDS Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL. SR$ABSDS absolutely disables an enabled rule, regardless of how many times the rule has been enabled. Compare with SR$DSABL.

SR$ADDB Add a rule to the beginning of a search list or before a specified rule.

SR$ADDE Add a rule to the end of a search list or after a specified rule.

SR$CREAT Create a search list.

SR$DEL	Delete a search list.
SR$DSABL	Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL. SR$DSABL disables a single SR$ENABL operation. Compare with SR$ABSDS.
SR$ENABL	Enable an optional search rule. Enabled rules can be disabled using SR$DSABL or SR$ABSDS.
SR$EXSTR	Determine if a search rule exists.
SR$FR_LS	Free list structure space allocated by SR$LIST or SR$READ.
SR$INIT	Initialize all search lists to system defaults.
SR$LIST	Return the names of all defined search lists.
SR$NEXTR	Read the next rule from a search list.
SR$READ	Read all of the rules in a search list.
SR$REM	Remove a search rule from a search list.
SR$SETL	Set the locator pointer for a search rule.
SR$SSR	Set a search list using a user–defined search rules file.

Some PRIMOS search rule subroutines require data types not available in all languages. All search rule subroutines can be executed using PL/I. All subroutine arguments are mandatory. Most arguments, such as *list_name*, are case–insensitive. However, arguments that compare a search rules value to an existing search rule are case–sensitive. Arguments cannot perform wildcard operations.

OPSR$

OPSR$ locates a file using a search list and open the file. This subroutine can also be used to create a file if the file sought does not exist.

Usage

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

```
CALL OPSR$ (list_name, referencing_dir, valid_types,
           action+newfile+getu, object_name, funit, type,
           found_path, code);
```

Parameters

list_name

INPUT. The name of the search list that OPSR$ should use to locate the desired file. If you set *list_name* to null, OPSR$ treats the *object_name* as a full pathname.

referencing_dir

INPUT. A search rule to substitute for the [referencing_dir] keywords in the search list. You establish either a search rules string or a null value for this argument. The search rule you specify here is temporarily substituted into the search list; then the search operation is performed on this search list. This substitute value is only kept for the duration of the subroutine call. If this argument is set to the null value, search rules containing the [referencing_dir] keyword are skipped over during the search operation.

valid_types

INPUT. Type of file system object to be located. The following values are permitted:

K$UNKN	Unknown file type, any file system object acceptable.
K$ACAT	Access categories (ACATs) only. OPSR$ can only verify the existence of an ACAT; OPSR$ does not open ACATs.
K$FILE	Files only.
K$SDIR	Segment directories only.
K$DIR	Dirctories only.

You can concatenate multiple *valid_types* options using a plus sign (+). For example, K$FILE+K$DIR can open either a file or a directory.

action

INPUT. Type of action to perform on the file system object when located. The following values are permitted:

K$EXST	Verify existence of *object_name*. This is the only value permitted for ACATs.
K$READ	Open *object_name* for reading.
K$WRIT	Open *object_name* for writing.
K$RDWR	Open *object_name* for update (reading and writing).

newfile

INPUT. If you are creating a new file, specify an *action* of K$WRIT or K$RDWR and then use *newfile* to specify the type of file you want to create. To specify *newfile*, use a plus sign (+) to concatenate the *action* argument with one of the following:

K$NSAM	New sequential access (SAM) file
K$NDAM	New direct access (DAM) file
K$NSGS	New sequential access (SAM) segment directory
K$NSGD	New direct access (DAM) segment directory

For example, to create a DAM file, you might specify K$WRIT+K$NDAM. *newfile* is an optional argument. If you do not specify *newfile* and circum– stances permit OPSR$ to create a new file, it creates a SAM file.

getu

INPUT. If you wish PRIMOS to automatically select the file unit number, use a plus sign to concatenate K$GETU to the *action* argument or the *newfile* argument (for example, K$WRIT+K$NDAM+ K$GETU). The *getu* argument is optional. If you omit *getu*, you must specify the file unit number using the *funit* argument.

object_name

INPUT. The name of the file system object for which you are searching. *object_name* can be either an objectname or a full pathname. If you supply an objectname, OPSR$ performs a search using *list_name*. If you supply a full pathname, OPSR$ locates the file system object without using *list_name*.

funit

INPUT. The file unit number that you wish to use for opening the file.

OUTPUT. If you specify a value of K$GETU in the *getu* argument, PRIMOS automatically assigns a file unit number to the file. The *funit* argument is then used to return the file unit number assigned by PRIMOS.

type

OUTPUT. The type of the object that OPSR$ successfully opened. Possible values are

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD top–level directory or subdirectory

found_path

OUTPUT. The absolute pathname of the file successfully opened.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$UIUS	The file has already been opened.
E$NRIT	You do not have read access rights to a file.
E$FNTF	The requested file cannot be located.
E$LIST	The search list specified cannot be located.

Discussion

OPSR$ is normally used to locate a file using a search list and then open the file. To use OPSR$ in this way, supply the filename to the *object_name* argument and the search list name to *list_name* argument.

OPSR$ can also be used to open a file without using a search list. To use OPSR$ in this way, supply the full pathname of the file to the *object_name* argument and a null value to *list_name.* A full pathname may include or omit the disk partition name. PRIMOS supplies an omitted partition name from the ATTACH$ search list (if one exists) or from the list of attached disks. Refer to the *Advanced Programmer's Guide II: File System* for further details on this use of ATTACH$.

OPSR$ can be used to create a new file, if no file of that name exists. To create a new file, you must set the *action* argument to K$WRIT or K$RDWR, and

OPSR$ must have sufficient information to determine where to create the file. For OPSR$ to create a file, either the *object_name* argument must contain the full pathname of the file, or the *object_name* argument must contain the name of the file and the *list_name* argument must be set to null. If *object_name* is a filename and *list_name* is null, OPSR$ creates the new file in the currently attached directory. The type of file created is determined by the value of the *newfile* argument. If you did not specify *newfile*, OPSR$ creates a SAM file.

The SRCH$$ subroutine can also be used to locate and open files. SRCH$$ has additional file access features not found in OPSR$; however, SRCH$$ cannot use the search rules facility to locate file system objects. If you wish to search for a file using both the search rules facility and a list of suffixes, use the OPSRS$ subroutine.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples locates and opens the file TESTFILE, using the MYLIST search list. Each example first inserts the search rule MYDIR>TOOLS into MYLIST at the location specified by [referencing_dir], and then searches MYLIST. TESTFILE may be a file or a segment directory. When OPSR$ locates TESTFILE, it opens it for update.

```
/*  Sample PL/I program for the OPSR$ subroutine */
OPEN_PROG: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
DCL OPSR$ ENTRY(CHAR(32) VAR,  CHAR(128) VAR,
                FIXED BIN,     FIXED BIN,
                CHAR(128) VAR, FIXED BIN,
                FIXED BIN,     CHAR(128) VAR,
                FIXED BIN);
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL REF CHAR(128) VAR STATIC INIT('MYDIR>TOOLS');
DCL V_TYPE FIXED BIN;
DCL KEYS FIXED BIN;
DCL OBJNAME CHAR(128) VAR STATIC INIT('TESTFILE');
DCL FUNIT FIXED BIN STATIC INIT('3');
DCL TYPE FIXED BIN;
DCL FOUND CHAR(128) VAR;
DCL CODE FIXED BIN;
CALL OPSR$  (LIST, REF, K$FILE+K$SDIR, K$RDWR,
             OBJNAME, FUNIT, TYPE, FOUND, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('File successfully opened: ', FOUND);
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the OPSR$ subroutine

$INSERT SYSCOM>KEYS.INS.FTN
C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 REFSIZE, REFPLUS(128)
      CHARACTER*128 REF
      INTEGER*2 V_TYPE
      INTEGER*2 KEYS
      INTEGER*2 OBJSIZE, OBJPLUS(128)
      CHARACTER*128 OBJNAME
      INTEGER*2 FUNIT
      INTEGER*2 TYPE
      INTEGER*2 FSIZE, FPLUS(128)
      CHARACTER*128 FOUND
      INTEGER*2 CODE
C  Record equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (REFSIZE, REFPLUS(1))
      EQUIVALENCE (REFPLUS(2), REF)
      EQUIVALENCE (OBJSIZE, OBJPLUS(1))
      EQUIVALENCE (OBJPLUS(2), OBJNAME)
      EQUIVALENCE (FSIZE, FPLUS(1))
      EQUIVALENCE (FPLUS(2), FOUND)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      REF(1:12) = 'MYDIR>TOOLS'
      REFSIZE = 12
      FUNIT = 3
      OBJNAME(1:8) = 'TESTFILE'
      OBJSIZE = 8
      FOUND = ''
C  Subroutine call
      CALL OPSR$(LPLUS, RPLUS, K$FILE+K$SDIR, K$RDWR,
     *  OBJPLUS, FUNIT, TYPE, FPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'File successfully opened: ', FOUND(1:FSIZE)
      CALL EXIT
C  Error routine
10    PRINT *, 'Error code: ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

OPSRS$

OPSRS$ locates and opens a file using a search list and a list of suffixes. This subroutine can also be used to create and open a file if the file sought does not exist. This subroutine is an extension of OPSR$. It provides support for suffix list checking in addition to the search rules support of OPSR$.

Usage

DCL OPSRS$ ENTRY (CHAR(32) VAR, CHAR(128) VAR, FIXED BIN,
FIXED BIN, CHAR(128) VAR, FIXED BIN,
FIXED BIN, FIXED BIN, PTR, CHAR(32) VAR,
FIXED BIN, CHAR(128) VAR, FIXED BIN);

CALL OPSRS$ (*list_name, referencing_dir, valid_types,*
action+null_suffix+newfile+getu, object_name, funit,
type, n_suffixes, suffix_list_ptr, basename, suffix_used,
***found_path, code*);**

Parameters

list_name

INPUT. The name of the search list that PRIMOS should use to locate the desired file. If you set *list_name* to null, OPSRS$ treats the *object_name* as a full pathname.

referencing_dir

INPUT. A search rule to substitute for the [referencing_dir] keywords in the search list. You establish either a search rules string or a null value for this argument. The search rule you specify here is substituted into the search list; then the search operation is performed on this modified search list. If you set this argument to the null value, search rules containing the [referencing_dir] keyword are skipped over during the search operation.

valid_types

INPUT. Type of file system object to be located. The following values are permitted:

K$UNKN	Unknown file type, any file system object acceptable.
K$ACAT	Access categories only. OPSRS$ can only verify the existence of an ACAT; OPSRS$ does not open ACATs.
K$FILE	Files only.

K$SDIR	Segment directories only.
K$DIR	Directories only.

You can concatenate multiple *valid_types* options using a plus sign (+). For example, K$FILE+K$DIR can open either a file or a directory.

action

INPUT. Type of action to perform on file system object when located. The following values are permitted:

K$EXST	Verify existence of *object_name*. This is the only value permitted for ACATs.
K$READ	Open *object_name* for reading.
K$WRIT	Open *object_name* for writing.
K$RDWR	Open *object_name* for update (reading and writing).

null_suffix

To search for a file with no suffix first, use a plus sign to concatenate K$NULF to the *action* argument (for example, K$WRIT+K$NULF). *null_suffix* is an optional argument. If you specify K$NULF, PRIMOS first searches for *object_name* with no suffix, then searches for *object_name* with the suffixes specified in the array pointed to by *suffix_list_ptr*. If you do not specify K$NULF, PRIMOS searches for *object_name* with no suffix last.

newfile

If you are creating a new file, use *action* to specify either K$WRIT or K$RDWR and then use *newfile* to specify what type of file to create. To specify *newfile*, use a plus sign (+) to concatenate the *action* argument with one of the following:

K$NSAM	New sequential access (SAM) file
K$NDAM	New direct access (DAM) file
K$NSGS	New sequential access (SAM) segment directory
K$NSGD	New direct access (DAM) segment directory

For example, to create a DAM file you might specify K$WRIT+ K$NDAM. *newfile* is an optional argument. If you do not specify *newfile*, PRIMOS creates a SAM file.

getu

If you wish PRIMOS to automatically select the file unit number, use a plus sign to concatenate K$GETU to the *action*, *null_suffix*, or *newfile* argument (for example, K$WRIT+K$NDAM+K$GETU). *getu* is an optional argument. If you do not specify *getu*, you must specify the file unit number using the *funit* argument.

object_name

INPUT. The name of the file system object for which you are searching. This name does not have to include the filename suffix. *object_name* can be a objectname or a full pathname. If you supply an objectname, OPSRS$ uses the search rules facility to perform a search using the *list_name* and a list of suffixes. If you supply a full pathname, OPSRS$ uses the list of suffixes to locate the file without using *list_name*.

funit

INPUT. The file unit number that you wish to use for opening the file.

OUTPUT. If you specify a value of K$GETU in the *getu* argument, PRIMOS automatically assigns a file unit number to the file. In that case, OPSRS$ uses *funit* to return the file unit number assigned by PRIMOS.

type

OUTPUT. The type of the object that OPSRS$ successfully accessed. Possible values are

0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	UFD top–level directory or subdirectory

n_suffixes

INPUT. The number of suffixes in the suffix list. Each suffix is an element in an array pointed to by the *suffix_list_ptr*. Set *n_suffixes* to 0 if no suffix checking is desired.

suffix_list_ptr

INPUT. A pointer to an array that contains a list of suffixes.

basename

OUTPUT. The filename of the successfully accessed file. The *basename* does not include the suffix (if any) of the filename.

suffix_used

OUTPUT. The sequence number of the suffix used to locate the file. The suffixes listed in the array are assigned sequential numbers, beginning with 1. A *suffix_used* value of 0 indicates that the file located had no suffix.

found_path

OUTPUT. The absolute pathname of the successfully opened file.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$UIUS	The file has already been opened.
E$NRIT	You do not have read access rights to a file.
E$FNTF	The requested file cannot be located.
E$LIST	The search list specified cannot be located.

Discussion

OPSRS$ performs all of the operations performed by OPSR$. It uses the search list you specify in *list_name* to locate and open the file system object you specify in *object_name*. However, OPSRS$ does not require that *object_name* include the filename suffix. Instead, OPSRS$ uses a list of suffixes when searching for a file.

OPSRS$ searches each rule in the search list for the specified filename plus the first listed suffix, then the second suffix, and so on. If you specify K$NULF, OPSRS$ first checks for *object_name* with no suffix; otherwise, OPSRS$ checks for *object_name* with no suffix after testing a search rule for the combination of *object_name* and all listed suffixes. If unsuccessful, OPSRS$ proceeds to the next rule in the search list and repeats this suffix–checking search operation.

If the *object_name* you supply already has a suffix, such as MYFILE.RUN, OPSRS$ appends suffixes from the list to *object_name*, creating filenames with multiple suffixes, such as MYFILE.RUN.CPL. However, OPSRS$ does not append a suffix to an identical suffix, (for example, MYFILE.RUN.RUN), but instead tests the *object_name* (MYFILE.RUN) without the duplicate suffix.

Creating a Suffix List: Declare a suffix list as an array of elements pointed to by the *suffix_list_ptr*. Elements are declared as CHAR(32) VAR. The number of elements should be equal to the value of the *n_suffixes* argument. The *n_suffixes* argument, not the number of elements in the array, determines how many suffixes are used for suffix checking.

Initialize this array with the suffixes to be used for suffix checking. Each suffix should begin with a period (.). User–defined suffixes, such as .MYSTUFF, and multiple suffixes, such as .MYSTUFF.CPL, are permitted.

You can also use the SRSFX$ subroutine to locate and open files using a suffix list. SRSFX$ has additional file access features not found in OPSRS$; however, SRSFX$ cannot use the search rules facility to locate file system objects.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples locates and opens the file TESTFILE, using the MYLIST search list and a list of suffixes. Using the suffix list, OPSRS$ locates either TESTFILE.CPL, TESTFILE.F77 or TESTFILE and opens it for update.

```
/*  Sample PL/I program for the OPSRS$ subroutine  */

OPEN_PROG: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
/*  Declarations                                  */
DCL OPSRS$ ENTRY(CHAR(32) VAR,  CHAR(128) VAR,
                 FIXED BIN,     FIXED BIN,
                 CHAR(128) VAR, FIXED BIN,
                 FIXED BIN,     FIXED BIN,
                 PTR,           CHAR(32) VAR,
                 FIXED BIN,     CHAR(128) VAR,
                 FIXED BIN);
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL REF CHAR(128) VAR STATIC INIT('');
DCL V_TYPE FIXED BIN;
DCL KEYS FIXED BIN;
DCL OBJNAME CHAR(128) VAR STATIC INIT('TESTFILE');
DCL FUNIT FIXED BIN STATIC INIT('3');
DCL TYPE FIXED BIN;
DCL N_SUFX FIXED BIN STATIC INIT('2');
DCL SUFX_PTR PTR;
DCL BASENAME CHAR(32) VAR;
DCL SUFX_USED FIXED BIN;
DCL FOUND CHAR(128) VAR;
DCL CODE FIXED BIN;
DCL SUFX_LIST(1:2) CHAR(32) VAR STATIC INIT
    ('.CPL', '.F77');
/*  Subroutine call                                  */
CALL OPSRS$(LIST, REF, K$FILE, K$RDWR, OBJNAME,
          FUNIT, TYPE, N_SUFX, ADDR(SUFX_LIST), BASENAME,
          SUFX_USED, FOUND, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('File successfully opened: ', FOUND);
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the OPSRS$ subroutine

$INSERT SYSCOM>KEYS.INS.FTN
C  Declarations of subroutine arguments
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 REFSIZE, REFPLUS(128)
      CHARACTER*128 REF
      INTEGER*2 V_TYPE
      INTEGER*2 KEYS
      INTEGER*2 OBJSIZE, OBJPLUS(128)
      CHARACTER*128 OBJNAME
      INTEGER*2 FUNIT
      INTEGER*2 TYPE
      INTEGER*2 N_SUFX
      INTEGER*4 SUFX_PTR
      INTEGER*2 BSIZE, BPLUS(32)
      CHARACTER*32 BASENAME
      INTEGER*2 SUFX_USED
      CHARACTER*128 FOUND
      INTEGER*2 CODE
C Declarations of suffix list
      INTEGER*2 SUFX_LIST(34)
      INTEGER*2 LSUF1, LSUF2
      CHARACTER*32 SUF1, SUF2
      INTEGER*2 FSIZE, FPLUS(128)
C  Define equivalences for character type arguments
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (REFSIZE, REFPLUS(1))
      EQUIVALENCE (REFPLUS(2), REF)
      EQUIVALENCE (OBJSIZE, OBJPLUS(1))
      EQUIVALENCE (OBJPLUS(2), OBJNAME)
      EQUIVALENCE (BSIZE, BPLUS(1))
      EQUIVALENCE (BPLUS(2), BASENAME)
      EQUIVALENCE (FSIZE, FPLUS(1))
      EQUIVALENCE (FPLUS(2), FOUND)
C  Define equivalences for suffix list
      EQUIVALENCE (LSUF1, SUFX_LIST(1))
      EQUIVALENCE (SUF1, SUFX_LIST(2))
      EQUIVALENCE (LSUF2, SUFX_LIST(18))
      EQUIVALENCE (SUF2, SUFX_LIST(19))
C  Assignments
         LIST(1:6) = 'MYLIST'
         LSIZE = 6
         REF(1:1) = ''
         REFSIZE = 1
         OBJNAME(1:8) = 'TESTFILE'
         OBJSIZE = 8
         FUNIT = 3
         N_SUFX = 2
         SUFX_PTR = LOC(SUFX_LIST(1))
         FOUND = ’’
         SUF1(1:4) = ’.CPL’
         LSUF1 = 4
         SUF2(1:4) = ’.F77’
         LSUF2 = 4
C  Subroutine call
    CALL OPSRS$(LPLUS, RPLUS, K$FILE, K$RDWR,
   *  OBJPLUS, FUNIT, TYPE, N_SUFX, SUFX_PTR, BPLUS,
   *  SUFX_USED, FPLUS, CODE)
    IF (CODE.NE.0) GO TO 10
    PRINT *, ‘File successfully opened: ’, FOUND(1:FSIZE)
    CALL CLOS$A(FUNIT)
    CALL EXIT
C  Error processing
10    PRINT *, ‘Error code: ’, CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$ABSDS
SR$ABS

SR$ABSDS disables an optional search rule in a specified search list. This subroutine absolutely disables an enabled rule, regardless of how many times the rule has been enabled. Compare with SR$DSABL.

Usage

```
DCL SR$ABSDS EXTERNAL ENTRY (CHAR(128) VAR,
                             CHAR(32) VAR, FIXED BIN);

CALL SR$ABSDS (rule, list_name, code);
```

Parameters

rule

INPUT. The search rule to be disabled. The rule specified in this argument should not include the -optional keyword. The rule specified in this argument should be otherwise identical to the rule in the corresponding search list. This argument is case-sensitive.

list_name

INPUT. The name of the search list in which the rule is located.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded. SR$ABSDS returns E$OK even if the rule was not enabled.
E$LIST	Search list does not exist.
E$RULE	Search rule cannot be located. Rule may be nonexistent or specified in the wrong case.
E$NTOP	Rule specified is not an optional rule.

Discussion

An optional search rule is a rule prefaced by the –optional keyword in the search rules file. Such rules are initially disabled when the search rules file is used to set the search list. PRIMOS ignores disabled search rules when performing a search operation. You can enable an optional search rule using SR$ENABL. SR$DSABL and SR$ABSDS are used to disable a rule that has been enabled using SR$ENABL.

Both SR$ENABL and SR$DSABL can be invoked repetitively for the same search rule. PRIMOS compares the number of calls to SR$ENABL with the number of calls to SR$DSABL. Each SR$DSABL call disables one invocation of SR$ENABL; therefore, to disable a rule you must invoke SR$DSABL as many times as SR$ENABL was called. If you use SR$DSABL to repeatedly disable a rule, you must invoke SR$ENABL a corresponding number of times to enable the rule.

SR$ABSDS absolutely disables a search rule. It reverses multiple calls to either SR$ENABL or SR$DSABL. If a rule is enabled, one invocation of SR$ABSDS disables the rule, regardless of how many times the rule had been enabled. If a rule is already disabled, one invocation of SR$ABSDS reverses any excess disable operations. A rule disabled by SR$ABSDS can be enabled by a single invocation of SR$ENABL.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples disables the search rule MYDIR>OPTTESTS in the search list MYLIST.

```
/*  Sample PL/I program for the SR$ABSDS subroutine */

ABS_SUB: PROCEDURE;
DCL SR$ABSDS EXTERNAL ENTRY(CHAR(128) VAR, CHAR(32) VAR,
                          FIXED BIN);
DCL RULE  CHAR(128) VAR STATIC INIT('MYDIR>OPTTESTS');
DCL LIST  CHAR(32) VAR STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$ABSDS(RULE, LIST, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Optional rule disabled');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the SR$ABSDS subroutine

C  Declarations
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      RULE(1:15) = 'MYDIR>OPTTESTS'
      RULESIZE = 15
C  Subroutine call
      CALL SR$ABSDS(RULEPLUS, LPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Optional rule disabled'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$ADDB
SR$ADB

SR$ADDB adds a search rule before an existing search rule in a search list. This subroutine can also be used to add a rule at the beginning of a search list.

Usage

```
DCL SR$ADDB EXTERNAL ENTRY (CHAR(32) VAR,
                            CHAR(128) VAR,
                            CHAR(128) VAR, FIXED BIN);

CALL SR$ADDB (list_name, old_rule, new_rule, code);
```

Parameters

list_name

INPUT. The name of the search list to which you wish to add a search rule.

old_rule

INPUT. An existing rule in the search list. SR$ADDB adds *new_rule* immediately before the rule specified in this argument. The value of *old_rule* must match an existing search rule; this argument is case–sensitive. The rule specified in this argument cannot be an administrator rule. To place a search rule at the beginning of a search list, specify a null string ( ) for this argument.

new_rule

INPUT. The search rule that you wish to add to the search list. This rule is added immediately before the rule specified in the *old_rule* argument. *new_rule* can be a pathname, an optional search rule, or a search rule keyword variable, but cannot be a –system or –insert keyword.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BPAR	Search rule to be added is invalid, for example, [garbage].
E$LIST	Specified search list does not exist.
E$RULE	The rule specified in *old_rule* cannot be located. Either the rule does not exist or it was specified in the wrong case.
E$ADMN	Attempting to add a rule before an administrator rule.

Discussion

SR$ADDB is used to add a single search rule before an existing search rule in a search list. It can also be used to add a single search rule at the beginning of a search list. To add a single search rule after an existing rule or at the end of a search list, use SR$ADDE. To append multiple search rules to an existing search list, use SR$SSR.

SR$ADDB cannot be used to add a rule before an administrator rule. It also cannot be used to add a rule that inserts multiple rules (such as the –system or –insert keywords). Use SR$SSR to add an –insert or –system keyword to an existing search list.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples adds the search rule [origin_dir] to the beginning of the MYLIST search list.

```
/*  Sample PL/I program for the SR$ADDB subroutine */

ADD_RULE: PROCEDURE OPTIONS(MAIN);
DCL SR$ADDB EXTERNAL ENTRY (CHAR(32) VAR, CHAR(128) VAR,
      CHAR(128) VAR, FIXED BIN);
DCL LIST CHAR(32) VARYING STATIC INIT('MYLIST');
DCL ORULE CHAR(128) VARYING STATIC INIT('');
DCL NRULE CHAR(128) VARYING STATIC INIT('[ORIGIN_DIR]');
DCL CODE  FIXED BIN;
CALL SR$ADDB(LIST, ORULE, NRULE, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Rule added to search list');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END ADD_RULE;
```

```
C  Sample FORTRAN 77 program for the SR$ADDB subroutine

C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 OSIZE, OPLUS(128)
      CHARACTER*128 ORULE
      INTEGER*2 NSIZE, NPLUS(128)
      CHARACTER*128 NRULE
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (OSIZE, OPLUS(1))
      EQUIVALENCE (OPLUS(2), ORULE)
      EQUIVALENCE (NSIZE, NPLUS(1))
      EQUIVALENCE (NPLUS(2), NRULE)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      ORULE(1:12) = ''
      OSIZE = 12
      NRULE(1:12) = '[ORIGIN_DIR]'
      NSIZE = 12
C  Subroutine call
      CALL SR$ADDB(LPLUS, OPLUS, NPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Rule added to search list'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$ADDE
SR$ADE

Adds a search rule to the end of a search list or adds a search rule after a specified rule in a search list.

Usage

```
DCL SR$ADDE EXTERNAL ENTRY (CHAR(32) VAR, CHAR(128)
                          VAR, CHAR(128) VAR, FIXED BIN);

CALL SR$ADDE (list_name, old_rule, new_rule, code);
```

Parameters

list_name

INPUT. The name of the search list to which you wish to add a search rule.

old_rule

INPUT. An existing rule in the search list. SR$ADDE adds the new rule immediately after the rule specified in this argument. The value of this argument must match an existing search rule; this argument is case–sensitive. The rule specified in this argument cannot be an administrator rule, unless it is the last administrator rule in the search list. To place a search rule at the end of a search list, specify a null string ( ) for this argument.

new_rule

INPUT. The search rule that you wish to add to the search list. This rule is added immediately after the rule specified in the *old_rule* argument. *new_rule* can be a pathname, an optional search rule, or an [origin_dir], [home_dir], or [referencing_dir] keyword. *new_rule* cannot be a –system or –insert keyword.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BPAR	Search rule to be added is invalid, for example, [garbage].
E$LIST	Specified search list does not exist.

E$RULE	The search rule specified in *old_rule* cannot be located. Either the rule does not exist or it was specified in the wrong case.
E$ADMN	Attempting to add a rule before an administrator rule.

Discussion

SR$ADDE is used to add a single search rule after an existing search rule in a search list. It can also be used to add a single search rule at the end of a search list. To add a single search rule before an existing rule or at the beginning of a search list, use SR$ADDB. To append multiple search rules to an existing search list, useSR$SSR. SR$SSR can also be used to add an –insert or –system keyword to an existing search list.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples adds the search rule MYDIR>TOOLS immediately after thesearch rule [origin_dir] in the MYLIST search list.

```
/*  Sample PL/I program for the SR$ADDE subroutine */

ADD_RULE: PROCEDURE OPTIONS(MAIN);
DCL SR$ADDE EXTERNAL ENTRY (CHAR(32) VAR, CHAR(128) VAR,
      CHAR(128) VAR, FIXED BIN);
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL ORULE CHAR(128) VAR STATIC INIT('[ORIGIN_DIR]');
DCL NRULE CHAR(128) VAR STATIC INIT('MYDIR>TOOLS');
DCL CODE  FIXED BIN;
CALL SR$ADDE(LIST, ORULE, NRULE, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Rule added to search list');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END ADD_RULE;
```

```
C  Sample FORTRAN 77 program for the SR$ADDE subroutine

C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 OSIZE, OPLUS(128)
      CHARACTER*128 ORULE
      INTEGER*2 NSIZE, NPLUS(128)
      CHARACTER*128 NRULE
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (OSIZE, OPLUS(1))
      EQUIVALENCE (OPLUS(2), ORULE)
      EQUIVALENCE (NSIZE, NPLUS(1))
      EQUIVALENCE (NPLUS(2), NRULE)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      ORULE(1:14) = '[ORIGIN_DIR]'
      OSIZE = 14
      NRULE(1:14) = 'MYDIR>TOOLS'
      NSIZE = 14
C  Subroutine call
      CALL SR$ADDE(LPLUS, OPLUS, NPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Rule added to search list'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$CREAT
SR$CRE

Creates a blank search list.

Usage

DCL SR$CREAT EXTERNAL ENTRY (CHAR(32) VAR, FIXED BIN);

CALL SR$CREAT (*list_name, code*);

Parameters

list_name

INPUT. The name of the search list that PRIMOS should create. A search list name should not exceed 22 characters.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$EXST	The search list specified already exists.
E$LIST	The search list name specified is an invalid name.

Discussion

SR$CREAT creates a blank search list; that is, a search list that does not contain any user–specified or system default search rules. This search list does, however, contain administrator rules if the System Administrator has established administrator rules for the search list.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples creates the MYLIST search list.

```
/*  Sample PL/I program for the SR$CREAT subroutine */

CREATE_SEARCH_LIST: PROCEDURE OPTIONS(MAIN);
DCL SR$CREAT EXTERNAL ENTRY (CHAR(32) VAR, FIXED BIN);
DCL LIST CHAR(32) VARYING STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$CREAT(LIST, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Search list created');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END CREATE_SEARCH_LIST;
```

```
C  Sample FORTRAN 77 program for the SR$CREAT subroutine

C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
C  Subroutine call
      CALL SR$CREAT(LPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Search list created'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$DEL

Deletes a specified search list.

Usage

DCL SR$DEL EXTERNAL ENTRY (CHAR(32) VAR, FIXED BIN);

CALL SR$DEL (*list_name, code*);

Parameters

list_name

INPUT. The name of the search list to be deleted.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$LIST	The specified list could not be located.

Discussion

SR$DEL completely deletes a search list. Both the user's search list and its contents (including administrator rules) are deleted. The search rules file that was used to set the search list is unaffected.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples deletes the MYLIST search list.

```
/*  Sample PL/I program for the SR$DEL subroutine */
DELETE_SEARCH_LIST: PROCEDURE OPTIONS(MAIN);
DCL SR$DEL EXTERNAL ENTRY (CHAR(32) VAR, FIXED BIN);
DCL LIST CHAR(32) VARYING STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$DEL(LIST, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Search list deleted');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END DELETE_SEARCH_LIST;
```

```
C  Sample FORTRAN 77 program for the SR$DEL subroutine

C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
C  Subroutine call
      CALL SR$DEL(LPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Search list deleted'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$DSABL
SR$DSA

SR$DSABL disables an optional search rule that was enabled by SR$ENABL. This subroutine reverses a single SR$ENABL operation. Compare with SR$ABSDS.

Usage

```
DCL SR$DSABL EXTERNAL ENTRY (CHAR(128) VAR,
                             CHAR(32) VAR, FIXED BIN);

CALL SR$DSABL (rule, list_name, code);
```

Parameters

rule

INPUT. The search rule to be disabled. The search rule should not include the –optional keyword. This argument is case–sensitive.

list_name

INPUT. The name of the search list in which the rule is located.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded. This return code does not indicate whether or not the rule is actually disabled. SR$DSABL returns E$OK if the rule is still enabled due to multiple nested SR$ENABL calls, or if the rule was never enabled.
E$LIST	Search list does not exist.
E$RULE	Search rule cannot be located. Rule may be nonexistent or specified in the wrong case.
E$NTOP	Rule specified is not an optional rule.

Discussion

SR$DSABL is used to disable an optional search rule in a search list. An optional search rule is a rule prefaced by the –optional keyword in the search rules file. Such rules are initially disabled when the search rules file is used to set the search list. PRIMOS ignores disabled search rules when performing a search operation on a search list. You can enable an optional search rule using SR$ENABL. SR$DSABL is used to disable a rule that has been enabled using SR$ENABL.

SR$ENABL can be invoked repetitively for the same search rule. PRIMOS compares the number of calls to SR$ENABL with the number of calls to SR$DSABL. Each SR$DSABL call disables one SR$ENABL call; therefore, to disable a rule, you must invoke SR$DSABL as many times as you invoked SR$ENABL.

You can also issue multiple SR$DSABL calls against a search rule, causing the search rule to be repetitively disabled. To enable such a rule, you must issue one more SR$ENABL call than the number of SR$DSABL calls you issued. A single call to SR$ABSDS reverses multiple SR$ENABL or SR$DSABL calls.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples disables the MYDIR>OPTTESTS search rule in the MYLIST search list.

```
/*  Sample PL/I program for the SR$DSABL subroutine */

DSABL_SUB: PROCEDURE;
DCL SR$DSABL EXTERNAL ENTRY(CHAR(128) VAR, CHAR(32) VAR,
                            FIXED BIN);
DCL RULE  CHAR(128) VAR STATIC INIT('MYDIR>OPTTESTS');
DCL LIST  CHAR(32) VAR STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$DSABL(RULE, LIST, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Optional rule disabled');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the SR$DSABL subroutine
C  Declarations
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      RULE(1:15) = 'MYDIR>OPTTESTS'
      RULESIZE = 15
C  Subroutine call
      CALL SR$DSABL(RULEPLUS, LPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Optional rule disabled'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$ENABL
SR$ENA

Enables an optional search rule. Enabled rules can be disabled using SR$DSABL or SR$ABSDS.

Usage

```
DCL SR$ENABL EXTERNAL ENTRY (CHAR(128) VAR,
                             CHAR(32) VAR, FIXED BIN);

CALL SR$ENABL (rule, list_name, code);
```

Parameters

rule

INPUT. The search rule to be enabled. The search rule specified here should be identical to an optional rule in the search list. The search rule specified in this argument should not include the –optional keyword. This argument is case–sensitive.

list_name

INPUT. The name of the search list in which the rule is located.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$LIST	Search list does not exist.
E$RULE	Search rule cannot be located. Rule may be nonexistent or specified in the wrong case.
E$NTOP	Rule specified is not an optional rule.

Discussion

An optional search rule is a rule prefaced by the –optional keyword in the search rules file. Such rules are initially disabled when the search rules file is used to set the search list. PRIMOS ignores disabled search rules when performing a search operation. When enabled, these search rules function as ordinary search rules. The same search rule can be repeatedly disabled and enabled. Only optional search rules can be disabled or enabled.

You can check for the existence of a disabled search rule using the SR$EXSTR subroutine and display disabled search rules using the SR$READ subroutine. Disabled search rules are not displayed by the SR$NEXTR subroutine or the LIST_SEARCH_RULES command. You can examine enabled search rules using any of these subroutines or the LIST_SEARCH_RULES command. An enabled search rule appears as an ordinary rule in a search list.

You use SR$ENABL to enable an optional search rule. You can use SR$DSABL or SR$ABSDS to disable a rule that has been enabled using SR$ENABL.

SR$ENABL calls can be nested; that is, your program can invoke SR$ENABL repetitively for the same search rule. SR$DSABL disables one invocation of SR$ENABL. To disable a rule you must call SR$DSABL as many times as SR$ENABL was called to enable the rule. SR$ABSDS absolutely disables an enabled search rule. That is, one invocation of SR$ABSDS disables the rule, regardless of how many times the rule had been enabled.

PRIMOS compares the number of SR$ENABL calls and SR$DSABL calls. You can issue multiple SR$DSABL calls against a disabled search rule. To enable a search rule disabled in this way, you must issue one more SR$ENABL call than the number of SR$DSABL calls you issued. A single call to SR$ABSDS reverses multiple SR$ENABL or SR$DSABL calls.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples enables the MYDIR>OPTTESTS search rule in the MYLIST search list.

```
/*  Sample PL/I program for the SR$ENABL subroutine */

ENABL_SUB: PROCEDURE;
DCL SR$ENABL EXTERNAL ENTRY(CHAR(128) VAR, CHAR(32) VAR,
                            FIXED BIN);
DCL RULE  CHAR(128) VAR STATIC INIT('MYDIR>OPTTESTS');
DCL LIST  CHAR(32) VAR STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$ENABL(RULE, LIST, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('Optional rule enabled');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the SR$ENABL subroutine

C  Declarations
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      RULE(1:15) = 'MYDIR>OPTTESTS'
      RULESIZE = 15
C  Subroutine call
      CALL SR$ENABL(RULEPLUS, LPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Optional rule enabled'
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$EXSTR
SR$EXS

Determines if a search rule exists in a specified search list.

Usage

```
DCL SR$EXSTR EXTERNAL ENTRY (CHAR(128) VAR, FIXED BIN,
                             CHAR(32) VAR, BIT(1) ALIGNED)
                             RETURNS(BIT(1) ALIGNED);
```

rule_exists = SR$EXSTR (*rule, rule_type, list_name, case_sensitive*);

Parameters

rule

INPUT. The search rule to be checked for existence in the specified search list.

rule_type

INPUT. Type of search rule to be checked. The following are available search rules types:

K$TEXT	Rule is an ordinary text string.
K$HMDR	Rule is the [home_dir] keyword.
K$ORDR	Rule is the [origin_dir] keyword.
K$RFDR	Rule is the [referencing_dir] keyword.
K$KEYW	Rule is a keyword that begins with a hyphen.
K$ANYTYPE	Rule can be either an ordinary text string or a keyword.

list_name

INPUT. The name of the search list that PRIMOS should search for the specified rule.

case_sensitive

INPUT. Specifies whether the comparison of *rule* and the rules in the search list should be case–sensitive or case–insensitive. '1'b specifies case–sensitive; '0'b specifies case–insensitive. If case–sensitive, the search rules mydir>test and MYDIR>TEST are different rules; if case–insensitive, these two search rules are equivalent.

rule_exists

RETURNED VALUE. Indicates the success or failure of the operation. '1'b indicates that the specified search rule was found in the search list. '0'b indicates that the search rule could not be found.

Discussion

SR$EXSTR determines whether a specified search rule exists in a search list. This search rule can be a pathname, an optional search rule, or a search rule keyword. This subroutine determines the existence of both disabled and enabled optional search rules.

When checking for the existence of a keyword, you must set both *rule* and *rule_type*:

- If the search rule sought is a keyword that begins with a hyphen, set *rule* to the keyword literal (including the hyphen) and *rule_type* to K$KEYW. Search rule keywords are not case–sensitive.
- If the search rule sought is [home_dir], [origin_dir], or [referencing_dir], set *rule* to null and *rule_type* to the type for that keyword.
- If the search rule sought combines a keyword variable and a partial pathname, such as [origin_dir]>TOOLS, set *rule* to the pathname portion of the search rule (in this case, *rules* = TOOLS), and set *rule_type* to the type for the keyword variable (in this case, *rule_type* = K$ORDR). The *rule* argument should not begin with an angle bracket (>).

SR$EXSTR can only check for a keyword as a literal; it cannot check for the current value assigned to a keyword. SR$EXSTR cannot locate the –insert or –system search rule keywords. Do not specify the –optional keyword when determining the existence of an optional search rule.

SR$EXSTR may indicate that a search rule is unlocatable for several reasons:

- The search list that you specified may not exist.
- The search rule may not exist in the search list specified.
- The search rule may be of a different type than the one specified in *rule_type*.

If you set the *case_sensitive* argument, the search rule that you specified in *rule* and the search rule in the search list may differ in case.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples checks for the existence of the MYDIR>TOOLS search rule in the MYLIST search list. The test is case–insensitive.

```
/*  Sample PL/I program for the SR$EXSTR subroutine */

EXIST_SUB: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
DCL SR$EXSTR EXTERNAL ENTRY(CHAR(128) VAR, FIXED BIN,
                           CHAR(32) VAR,
                           BIT(1) ALIGNED) RETURNS(BIT(1)
                           ALIGNED);
DCL RULE  CHAR(128) VAR STATIC INIT('MYDIR>TOOLS');
DCL TYPE  FIXED BIN;
DCL LIST  CHAR(32) VAR STATIC INIT('MYLIST');
DCL CASE  BIT(1) ALIGNED STATIC INIT('0'b);
DCL EXIST BIT(1) ALIGNED;
EXIST =  SR$EXSTR(RULE, K$TEXT, LIST, CASE);
PUT SKIP LIST('Existence of rule is: ', EXIST);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the SR$EXSTR subroutine
$INSERT SYSCOM>KEYS.INS.FTN
C  Declarations
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 TYPE
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 CASE
      INTEGER*2 EXIST
C  Equivalences
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      RULE(1:18) = 'MYDIR>TOOLS'
      RULESIZE = 18
      CASE = :000000
C  Subroutine call
      EXIST = SR$EXSTR(RULEPLUS, K$TEXT, LPLUS, CASE)
      IF (EXIST.EQ.0) GO TO 10
      PRINT *, 'Rule exists', EXIST
      CALL EXIT
10    PRINT *, 'Rule does not exist', EXIST
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$FR_LS
SR$FRL

Frees the space allocated to a linked list structure by SR$LIST or SR$READ.

Usage

DCL SR$FR_LS EXTERNAL ENTRY (PTR, FIXED BIN);

CALL SR$FR_LS (*structure_ptr, code*);

Parameters

structure_ptr

INPUT. A pointer to the structure to be freed. You set this pointer value using the value of the *output_ptr* argument of SR$LIST or SR$READ.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BDAT	Encountered invalid pointer.

Discussion

SR$FR_LS frees space allocated by the SR$LIST and SR$READ subroutines. You should invoke SR$FR_LS after every successful invocation of SR$LIST or SR$READ. If either SR$LIST or SR$READ fails (that is, returns a nonzero value for the *code* argument) no space is allocated, and SR$FR_LS does not need to be invoked.

SR$FR_LS deletes a structure by following the structure's internal pointers. It does not examine the contents of the other entry fields in the structure. If SR$FR_LS encounters an invalid pointer, it returns an E$BDAT error code. The subroutine may have already freed part of the linked list when it encountered the invalid pointer.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples frees storage space allocated by the SR$READ subroutine.

```
/*  Sample PL/I program for the SR$FR_LS subroutine */

FREE_LIST_STRUCTURE: PROCEDURE OPTIONS(MAIN);
DCL SR$FR_LS EXTERNAL ENTRY (PTR, FIXED BIN);
DCL LOC  PTR;
DCL CODE  FIXED BIN;
DCL SR$READ EXTERNAL ENTRY (FIXED BIN, CHAR(32) VAR, PTR,
                           FIXED BIN);
CALL SR$READ(VER, ‘MYLIST’, LOC, CODE);
.
.
.
CALL SR$FR_LS(LOC, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST(‘List structure space freed’);
ELSE
   PUT SKIP LIST(‘Error code: ’, CODE);
PUT SKIP;
END FREE_LIST_STRUCTURE;

C  Sample FORTRAN 77 program for the SR$FR_LS subroutine

C  Declarations
      INTEGER*4 PTR
      INTEGER*2 CODE
C  Create the structure to be freed
10    CALL SR$LIST(INTS(1), PTR, CODE)
      .
      .
      .
C  Call SR$FR_LS subroutine
20    CALL SR$FR_LS(PTR, CODE)
      IF (CODE.NE.0) GO TO 30
      PRINT *, ‘List structure space freed’
      CALL EXIT
C  Error processing
30    PRINT *, ‘Error code ’, CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$INIT
SR$INI

SR$INIT initializes all search lists to system defaults.

Usage

DCL SR$INIT EXTERNAL ENTRY (FIXED BIN);

CALL SR$INIT (*code*);

Parameters

code

OUTPUT. Standard error code. Because SR$INIT can initialize multiple search lists, multiple errors can occur. The returned error code indicates only the most recently encountered of these errors. Possible values are

E$OK	Operation succeeded. All search lists initialized.
E$FNTF	A system default file contains an –insert keyword that refers to a nonexistent file. One or more search lists have not been initialized. Search lists not in error have been initialized.
E$NEST	A system default file contains an –insert keyword that invokes a circular reference. One or more search lists have not been initialized. Search lists not in error have been initialized.

Discussion

SR$INIT initializes all of the user's search lists to system defaults. System default rules include all rules found in directory SEARCH_RULES*, including system rules and administrator rules. If no system defaults exist for a search list, SR$INIT deletes that search list. If an error occurs during initialization, SR$INIT sets the *code* argument and does not initialize the list in error; it proceeds to initialize to system defaults all lists that are not in error.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples initializes all of the user’s search lists. Search lists with system defaults are reset to default rules. Search lists without system defaults are deleted.

```
/*  Sample PL/I program for the SR$INIT subroutine */
INITIALIZE_SEARCH_LISTS: PROCEDURE OPTIONS(MAIN);
DCL SR$INIT EXTERNAL ENTRY (FIXED BIN);
DCL CODE  FIXED BIN;
CALL SR$INIT(CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST(‘Search lists initialized’);
ELSE
   PUT SKIP LIST(‘Error code: ’, CODE);
PUT SKIP;
END INITIALIZE_SEARCH_LISTS;
```

```
C  Sample FORTRAN 77 program for the SR$INIT subroutine

C  Declarations
      INTEGER*2 CODE
C  Subroutine call
      CALL SR$INIT(CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, ‘Search lists initialized’
      CALL EXIT
C  Error processing
10    PRINT *, ‘Error code ’, CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$LIST
SR$LIS

Returns the names of the user's search lists.

Usage

DCL SR$LIST EXTERNAL ENTRY (FIXED BIN, PTR, FIXED BIN);

CALL SR$LIST (*version, output_ptr, code*);

Parameters

version

INPUT. The version number of the requested structure. Different version numbers are assigned to structures with fields of differing lengths. For Rev. 21.0, set this argument to 1.

output_ptr

OUTPUT. Pointer to a structure used to hold the search list names. This structure contains one entry for each of the user's search lists. If the user has no search lists, this pointer is set to null. See Structure Description below.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BVER	Version number is invalid.

Structure Description

The parameter *output_ptr* points to a structure, *list_struc*, as follows:

```
DCL 1     list_struc,
          2       version FIXED BIN,
          2       length FIXED BIN,
          2       next PTR OPTIONS(SHORT),
          2       list_name CHAR(32) VAR,
          2       template CHAR(128) VAR;
```

version

INPUT. The version number of the structure (for Rev. 21.0 and later revisions, the version number is always 1).

length

INPUT. The length of a structure entry (always 172 bytes).

next

OUTPUT. A pointer to the next entry. If this is the last entry, the value is null.

list_name

OUTPUT. The name of the search list.

template

OUTPUT. The pathname of the search rules file used to set the search list. Only one pathname is listed, even if multiple search rule files were used to set the search list. If the search list contains system rules and administrator rules, *template* is the search rule file for the system rules. If the search list contains user–specified rules, *template* is the user's search rules file supplied to SR$SSR or the SET_SEARCH_RULES command.

Discussion

SR$LIST copies information about all of the user's search lists into a user–specified structure. SR$LIST creates a separate structure entry for each of the user's search lists.

It is the user's responsibility to free the space allocated for the structure used by SR$LIST. This space can be freed using the SR$FR_LS subroutine.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples creates the structure LISTSTRUC and copies into it a separate entry for the name of each of the user's search lists.

```
/*  Sample PL/I program for the SR$LIST subroutine */
LIST_NAMES: PROCEDURE;
DCL SR$LIST EXTERNAL ENTRY(FIXED BIN, PTR, FIXED BIN);
DCL VER   FIXED BIN STATIC INIT('1');
DCL LOC   PTR;
DCL CODE  FIXED BIN;
DCL 1 LISTSTRUC BASED(LOC),
     2 VERSION FIXED BIN,
     2 LENGTH FIXED BIN,
     2 NEXT PTR OPTIONS(SHORT),
     2 LIST CHAR(32) VAR,
     2 TEMPLATE CHAR(128) VAR;
CALL SR$LIST(VER, LOC, CODE);
```

```
IF (CODE = 0)
THEN BEGIN;
       DO WHILE (LOC ^= NULL());
       PUT SKIP LIST('List name: ', LIST);
       PUT SKIP LIST('Search rules file: ', TEMPLATE);
       LOC = NEXT;
       END;
     END;
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END;


C  Sample FORTRAN 77 program for the SR$LIST subroutine

C  Declarations
      INTEGER*4 PTR, NPTR, PTR1
      INTEGER*2 CODE, LISTL, FILEL
C  Establish space for output structure
      INTEGER*2 LISTSTRUC(86)
      CHARACTER*32 LIST
      CHARACTER*128 FILE
C  Redefine the structure entries
      EQUIVALENCE (NPTR, LISTSTRUC(3))
      EQUIVALENCE (LISTL, LISTSTRUC(5)),
                  (LIST, LISTSTRUC(6))
      EQUIVALENCE (FILEL, LISTSTRUC(22)),
                 (FILE, LISTSTRUC(23))
C  Subroutine call
      CALL SR$LIST(INTS(1), PTR, CODE)
      IF (CODE.NE.0) GO TO 30
      PTR1 = PTR
C  Keep analyzing until the pointer is null
10    IF (AND(PTR,:1777600000).EQ.:1777600000) GO TO 20
C  Copy the structure to place where we can access it
      CALL MOVEW$(PTR, LOC(LISTSTRUC), INTS(86))
      PRINT *, 'List name: ', LIST(1:LISTL)
      PRINT *, 'Search rules file: ', FILE(1:FILEL)
      PRINT *
      PTR = NPTR
      GO TO 10
C  Normal exit
20    CALL SR$FR_LS(PTR1, CODE)
      IF (CODE.NE.0) GO TO 30
      CALL EXIT
C  Error processing
30    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$NEXTR
SR$NEX

Reads the next rule from a search list.

Usage

```
DCL SR$NEXTR EXTERNAL ENTRY (CHAR(32) VAR,
                    FIXED BIN(31), CHAR(128) VAR, PTR,
                    FIXED BIN, CHAR(128) VAR,
                    FIXED BIN) RETURNS (FIXED BIN(31));
```

curr_rule_handle **= SR$NEXTR (*list_name, prev_rule_handle, referencing_dir, locator, rule_type, rule, code*);**

Parameters

list_name

INPUT. The name of the search list containing the rules to be read.

prev_rule_handle

INPUT. The point in the search list at which to start reading. Use the value K$BGN to read the first rule in the search list. To read other rules in the search list, use the value of the *curr_rule_handle* argument from a previous invocation of SR$NEXTR.

referencing_dir

INPUT. A search rule to substitute for the [referencing_dir] keywords in the search list. You establish either a search rules string or the null value for this argument. The search rule that you specify is substituted into the search list; then the read operation is performed on this modified search list. If you specify the null value, SR$NEXTR skips over any search rule containing the [referencing_dir] keyword and reads the next rule in the search list. The value you establish for [referencing_dir] keywords only applies to the current invocation of SR$NEXTR.

locator

OUTPUT. This argument reads the locator value established for the search rule. PRIMOS sets the locator values for search rules in the ENTRY$ search list. You can use the SR$SETL subroutine to set locator values for rules in user–defined search lists and the ENTRY$ search list. Locators are not set for other search lists. If a locator value for a rule is not set, this argument defaults to null.

rule_type

OUTPUT. The type of search rule read. Possible values are

K$TEXT	1	Rule is an ordinary text string.
K$HMDR	2	Rule is the [home_dir] keyword.
K$ORDR	3	Rule is the [origin_dir] keyword.
K$RFDR	4	Rule is the [referencing_dir] keyword.
K$KEYW	8	Rule is a keyword that begins with a hyphen.

rule

OUTPUT. The search rule read by this operation. If the search rule in the list is [origin_dir], *rule* returns the name of the origin directory. If the search rule in the list is [home_dir], *rule* returns an asterisk (*). If the search rule in the list is [referencing_dir], *rule* returns the pathname supplied by the *referencing_dir* argument. SR$NEXTR skips over [referencing_dir] rules that are set to the null value and disabled optional search rules. If the search rule is some other keyword (such as –added_disks), *rule* returns the keyword itself.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$LIST	Search list specified does not exist.
E$EOL	Attempting to read beyond the end of the list.

curr_rule_handle

RETURNED VALUE. The internal handle of the rule read by this operation. To read the next rule, you input this *curr_rule_handle* value to the *prev_rule_handle* for the next invocation of SR$NEXTR. If SR$NEXTR is invoked when there are no more rules in the list, this argument is set to K$END.

Discussion

SR$NEXTR is used to sequentially read the rules in a search list, one rule at a time. Each invocation of SR$NEXTR reads one rule. To read all of the rules in a search list in one operation, use SR$READ.

Usually, SR$NEXTR is invoked in a program loop, in which the first invocation reads the first rule in the search list and returns its value and address. The next invocation of SR$NEXT uses this output address as its input, and returns the second search rule's value and address. Each invocation takes the value of *curr_rule_handle* from the previous call to SR$NEXTR and uses that as the *prev_rule_handle* input.

SR$NEXTR reads locator pointer values. To set a locator pointer value, you use SR$NEXTR to supply the address of a search rule to the SR$SETL subroutine. For further details, refer to SR$SETL.

SR$NEXTR does not read disabled optional search rules. It reads enabled optional search rules as ordinary search rules with no indication that these rules are optional. SR$READ does read disabled optional search rules. For further details on optional search rules refer to the SR$ENABL subroutine.

If you call SR$NEXTR when there are no more search rules to read in the search list, the *rule* argument returns the value of the last rule in the list (the previous rule), the *code* argument returns a value of E$EOL, and the *curr_rule_handle* returns K$END.

The SR$NEXTR *curr_rule_handle* is a required input parameter for the SR$SETL subroutine.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples sequentially reads the search rules in the MYLIST search list. Each invocation of SR$NEXTR reads one search rule. Each example supplies a value to the [referencing_dir] search rule keyword.

```
/*  Sample PL/I program for the SR$NEXTR subroutine */

NEXT_SUB: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
DCL SR$NEXTR EXTERNAL ENTRY(CHAR(32) VAR, FIXED BIN(31),
                  CHAR(128) VAR, PTR, FIXED BIN,
                  CHAR(128) VAR, FIXED BIN,
                  RETURNS (FIXED BIN(31));
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL PREV  FIXED BIN(31);
DCL REFD  CHAR(128) VAR STATIC INIT('MYDIR>TOOLS');
DCL LOC   PTR;
DCL 1 LOCATOR DEFINED (LOC),
  2 FAULT BIT(1),
  2 RING BIT(2),
  2 FMT BIT(1),
  2 SEGNO BIT(12),
  2 WORD BIT(16),
  2 OFFSET BIT(4),
  2 RES BIT(12);
DCL RTYPE FIXED BIN;
DCL RULE  CHAR(128) VAR;
DCL CODE  FIXED BIN;
DCL CURR  FIXED BIN(31);
DCL X     FIXED BIN;
CURR = SR$NEXTR(LIST, K$BGN, REFD, LOC, RTYPE, RULE,
               CODE);
IF (CODE = 0)
THEN
   BEGIN;
   PUT SKIP LIST('The first rule is: ', RULE);
   PUT SKIP LIST('Locator seg no: ',LOCATOR.SEGNO);
   PUT SKIP LIST('Locator word no: ',LOCATOR.WORD);
   END;
ELSE GO TO A;
PUT SKIP;
  DO X = 1 TO 10;
    CURR = SR$NEXTR(LIST, CURR, REFD, LOC, RTYPE, RULE,
                    CODE);
    IF (CODE = 0)
    THEN
       BEGIN;
       PUT SKIP LIST('The rule is: ', RULE);
       PUT SKIP LIST('Locator seg no: ',LOCATOR.SEGNO);
       PUT SKIP LIST('Locator word no: ',LOCATOR.WORD);
       END;
    ELSE  GO TO A;
  END;
A:  PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END NEXT_SUB;
```

```
C  Sample FORTRAN 77 program for the SR$NEXTR subroutine

$INSERT SYSCOM>KEYS.INS.FTN
C  Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*4 PREV
      INTEGER*2 REFSIZE, REFPLUS(128)
      CHARACTER*128 REF
      INTEGER*4 PTR
      INTEGER*2 TYPE
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 CODE
      INTEGER*2 RETVAL
C  Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (REFSIZE, REFPLUS(1))
      EQUIVALENCE (REFPLUS(2), REF)
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      REF(1:15) = ''
      REFSIZE = 15
      RULE = ''
C  Subroutine call
      RETVAL = SR$NEXTR(LPLUS, K$BGN, REFPLUS, PTR,
     *  TYPE, RULEPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'The first rule is:', RULE
      CALL EXIT
C  Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$READ
SR$REA

Reads the rules in a specified search list into a structure established by the user. SR$READ reads all rules, including disabled rules.

Usage

```
DCL SR$READ EXTERNAL ENTRY (FIXED BIN, CHAR(32) VAR,
                                        PTR, FIXED BIN);
```

CALL SR$READ (*version, list_name, output_ptr, code*);

Parameters

version

INPUT. The version number of the requested structure. Different version numbers are assigned to structures with fields of differing lengths. For Rev. 21.0, set this argument to 1.

list_name

INPUT. The name of the search list to be read.

output_ptr

OUTPUT. A pointer to the structure that contains the rules copied from the search list. If the specified search list contains no rules, this pointer is set to null. See Structure Description below.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BVER	Version number is invalid.
E$LIST	Search list specified does not exist.

Structure Description

The parameter *output_ptr* points to a structure, *rules_struc*, as follows:

```
DCL 1    rules_struc,
         2 version FIXED BIN,
         2 length FIXED BIN,
         2 next PTR OPTIONS(SHORT),
         2 rule CHAR(128) VAR,
         2 enabled BIT(1) ALIGNED;
```

version

INPUT. The version number of the structure (for Rev. 21, the version number is always 1).

length

INPUT. The length of a structure entry (always 140 bytes).

next

OUTPUT. A pointer to the next entry. If the current entry is the last entry in the structure, *next* is set to null.

rule

OUTPUT. The search rule itself.

enabled

OUTPUT. An indicator of whether or not the rule is enabled. A value of '1'b indicates either an ordinary search rule or an enabled optional search rule. A value of '0'b indicates a disabled optional search rule.

Discussion

SR$READ copies all of the search rules in a user's search list into a user–specified structure. The search list itself is unaffected by this copy operation. SR$READ creates a separate structure entry for each search rule. To check for the existence of an individual search rule, use SR$EXSTR; to read an individual search rule, use SR$NEXTR.

SR$READ reads disabled optional search rules. Optional search rules are disabled when they are initially set in a search list. Disabled optional search rules are not shown by the LIST_SEARCH_RULES command or by the SR$NEXTR read operation. For further details on creating optional search rules, refer to the *Advanced Programmer's Guide II: File System*. For further details on enabling optional search rules, refer to the SR$ENABL subroutine.

It is the user's responsibility to free the space allocated for the structure used by SR$LIST. This space can be freed using the SR$FR_LS subroutine.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples sequentially reads all of the search rules in search list MYLIST into the structure READSTRUC. Each READSTRUC entry contains information about one search rule.

```
/*  Sample PL/I program for the SR$READ subroutine */

READ_SUB: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
DCL RETVAL FIXED BIN(31);
DCL SR$READ EXTERNAL ENTRY(FIXED BIN, CHAR(32) VAR,
                           PTR, FIXED BIN);
DCL VER   FIXED BIN STATIC INIT('1');
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL LOC   PTR;
DCL CODE  FIXED BIN;
DCL 1 READSTRUC BASED(LOC),
     2 VERSION FIXED BIN,
     2 LENGTH FIXED BIN,
     2 NEXT PTR OPTIONS(SHORT),
     2 RULE_STR CHAR(128) VAR,
     2 ENABLED BIT(1) ALIGNED;
CALL SR$READ(VER, LIST, LOC, CODE);
IF (CODE = 0)
THEN BEGIN;
      DO WHILE (LOC ^= NULL());
      PUT SKIP LIST(‘The rule is: ', RULE_STR);
      IF (ENABLED = '1'b) THEN PUT SKIP LIST(‘Rule is enabled');
                          ELSE PUT SKIP LIST(‘Rule is disabled');
      LOC = NEXT;
      END;
     END;
ELSE
   PUT SKIP LIST(‘Error code: ', CODE);
PUT SKIP;
END;
```

```
C  Sample FORTRAN 77 program for the SR$READ subroutine

C  Declarations
      INTEGER*4 PTR, NPTR, PTR1
      INTEGER*2 CODE, RULEL
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
C  Establish space for output structure
      INTEGER*2 STRUCT(70)
      CHARACTER*128 RULE
C  Redefine the structure entries
      EQUIVALENCE (NPTR, STRUCT(3))
      EQUIVALENCE (RULEL, STRUCT(5)), (RULE, STRUCT(6))
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
C  Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
C  Subroutine call
      CALL SR$READ(INTS(1), LPLUS, PTR, CODE)
      IF (CODE.NE.0) GO TO 30
      PTR1 = PTR
C  Keep analyzing until the pointer is null
10    IF (AND(PTR,:1777600000).EQ.:1777600000) GO TO 20
C  Copy the structure to place where we can access it
      CALL MOVEW$(PTR, LOC(STRUCT), INTS(70))
      PRINT *, 'The rule is: ', RULE(1:RULEL)
      PRINT *
      PTR = NPTR
      GO TO 10
C  Normal exit
20    CALL SR$FR_LS(PTR1, CODE)
      IF (CODE.NE.0) GO TO 30
      CALL EXIT
C  Error processing
30    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$REM

Removes a search rule from a specified search list.

Usage

```
DCL SR$REM EXTERNAL ENTRY (CHAR(32) VAR, CHAR(128)
                                    VAR, FIXED BIN);

CALL SR$REM (list_name, rule, code);
```

Parameters

list_name

INPUT. The name of the search list from which a search rule is to be removed.

rule

INPUT. The search rule to be removed from the list. The value you specify for *rule* must be in the same case (uppercase or lowercase letters) as the corresponding search rule in the search list.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$LIST	Search list does not exist.
E$RULE	Rule cannot be found in search list specified. Rule may be nonexistent or in the wrong case.
E$ADMN	Rule specified for removal is an administrator rule.

Discussion

SR$REM removes the first instance of a search rule that matches the value of the SR$REM *rule* argument. This matching operation is case–sensitive. SR$REM can delete user–specified and system default search rules and keywords. SR$REM cannot delete administrator search rules.

You can use SR$REM to remove search rule keywords, for example, [home_dir] and –added_disks. You remove a keyword variable by specifying the keyword, not by specifying the current value of that keyword variable.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples removes the search rule MYDIR>TESTS from the MYLIST search list.

```
/*  Sample PL/I program for the SR$REM subroutine */

REMOVE_RULE: PROCEDURE OPTIONS(MAIN);
DCL SR$REM EXTERNAL ENTRY (CHAR(32) VAR, CHAR(128) VAR,
                               FIXED BIN);
DCL RULE CHAR(128) VAR STATIC INIT('MYDIR>TESTS');
DCL LIST CHAR(32) VAR STATIC INIT('MYLIST');
DCL CODE  FIXED BIN;
CALL SR$REM(LIST, RULE, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST('The rule has been removed');
ELSE
   PUT SKIP LIST('Error code: ', CODE);
PUT SKIP;
END REMOVE_RULE;
```

```
C   Sample FORTRAN 77 program for the SR$REM subroutine

C   Declarations
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 RULESIZE, RULEPLUS(128)
      CHARACTER*128 RULE
      INTEGER*2 CODE
C   Equivalences
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (RULESIZE, RULEPLUS(1))
      EQUIVALENCE (RULEPLUS(2), RULE)
C   Assignments
      LIST(1:6) = 'MYLIST'
      LSIZE = 6
      RULE(1:12) = 'MYDIR>TESTS'
      RULESIZE = 12
C   Subroutine call
      CALL SR$REM(LPLUS, RULEPLUS, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, 'Rule removed from list'
      CALL EXIT
C   Error processing
10    PRINT *, 'Error code ', CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$SETL
SR$SET

Modifies the locator pointer of a search rule.

Usage

```
DCL SR$SETL EXTERNAL ENTRY (FIXED BIN(31), PTR,
                                        FIXED BIN);
```

CALL SR$SETL (*rule_handle, locator, code*);

Parameters

rule_handle

INPUT. The handle you use to locate the rule to be modified. You obtain the *rule_handle* value from the *curr_rule_handle* argument returned by the SR$NEXTR subroutine.

locator

INPUT. The value you wish to establish for the locator pointer. A locator pointer value should be a valid address in memory. You can set this argument to null to delete a previous locator pointer value.

code

OUTPUT. Standard error code. Possible values are

E$OK	Operation succeeded.
E$BPAR	*rule_handle* is set to a null address.
E$ADMN	Attempted to set the locator pointer of an administrator rule.

Discussion

SR$SETL is used to set the locator pointer for a rule. Each search rule in the ENTRY$ search list and in user–defined search lists has a locator pointer. When the search list is set, these locator pointers are initialized to null values. If the locator is null, PRIMOS searches for a file system object by searching the file system. Once the location of the file system object is known, the locator pointer can be assigned the address in memory of that object. If the locator is not null, PRIMOS locates the file system object by going to the address in memory specified by the locator pointer. Assigning a locator pointer value speeds subsequent use of a search rule.

PRIMOS automatically assigns locator pointer values to the search rules in the ENTRY$ search list. The first search operation that uses an ENTRY$ search rule causes PRIMOS to set that search rule's locator pointer. Using SR$SETL, you can set locator pointers of search rules in user–defined search lists and search rules in the ENTRY$ search list.

SR$SETL is used in combination with SR$NEXTR. You first read the search rule using SR$NEXTR. SR$NEXTR returns an address that you supply as the *rule_handle* argument input to SR$SETL. After setting the locator pointer, you can check this value using SR$NEXTR. The SR$NEXTR *locator* argument displays the locator pointer value.

Locator pointer values are not used by ATTACH$, BINARY$, COMMAND$, or INCLUDE$ search list processing. You cannot use SR$SETL to set a locator value for an administrator rule.

Example

The following example sets the locator pointer of the first search rule in the MYLIST search list. It first calls SR$NEXTR to return the address of the search rule. It then supplies this search rule address and a locator pointer value to SR$SETL. Finally, it calls SR$NEXTR again to confirm the new locator pointer value for that search rule.

```
/*  Sample PL/I program for the SR$SETL subroutine */

SET_LOCATOR: PROCEDURE OPTIONS(MAIN);
%INCLUDE 'SYSCOM>KEYS.PL1';
DCL SR$NEXTR EXTERNAL ENTRY(CHAR(32) VAR, FIXED BIN(31),
                           CHAR(128) VAR, PTR,
                           FIXED BIN, CHAR(128) VAR,
                           FIXED BIN) RETURNS (FIXED
                           BIN(31));
DCL LNAME CHAR(32) VAR STATIC INIT('MYLIST');
DCL PREV  FIXED BIN(31);
DCL REFD  CHAR(128) VAR STATIC INIT('');
DCL LOC   PTR;
DCL 1 LOCATOR DEFINED (LOC),
      2 FAULT BIT(1),
      2 RING BIT(2),
      2 FORMAT BIT(1),
      2 SEGNO BIT(12),
      2 WORDNO BIT(16);
DCL RTYPE FIXED BIN;
DCL RULE  CHAR(128) VAR;
DCL CODE  FIXED BIN;
DCL RETVAL FIXED BIN(31);
DCL SR$SETL EXTERNAL ENTRY (FIXED BIN(31), PTR, FIXED BIN);
DCL LOC2   PTR;
DCL 1 LOCATOR2 DEFINED (LOC2),
      2 FAULT2 BIT(1),
      2 RING2 BIT(2),
      2 FORMAT2 BIT(1),
      2 SEGNO2 BIT(12),
      2 WORDNO2 BIT(16);
SEGNO2 = '100111111111'b;
WORDNO2 = '0101010101010101'b;
/*   Perform the calls   */
RETVAL = SR$NEXTR(LNAME, K$BGN, REFD, LOC, RTYPE, RULE, CODE);
  IF (CODE = 0) THEN
    BEGIN;
    PUT SKIP LIST('The rule is: ', RULE);
    PUT SKIP LIST('The original segment number is:',LOCATOR.SEGNO);
    PUT SKIP LIST('The original word number is : ',LOCATOR.WORDNO);
    END;
  ELSE GO TO A;
CALL SR$SETL(RETVAL, LOC2, CODE);
  IF (CODE = 0) THEN
    PUT SKIP LIST('Locator pointer set');
  ELSE GO TO A;
RETVAL = SR$NEXTR(LNAME, K$BGN, REFD, LOC, RTYPE, RULE, CODE);
  IF (CODE = 0) THEN
    BEGIN;
    PUT SKIP LIST('The rule is: ', RULE);
    PUT SKIP LIST('The reset segment number is: ',LOCATOR.SEGNO);
    PUT SKIP LIST('The reset word number is: ',LOCATOR.WORDNO);
    END;
  ELSE GO TO A;
A:  PUT SKIP LIST('Error code is: ', CODE);
END SET_LOCATOR;
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

SR$SSR

SR$SSR sets a search list using a user–defined search rules file. This subroutine can create a new search list, overwrite an existing search list, or append rules to an existing search list.

Usage

```
DCL SR$SSR EXTERNAL ENTRY (CHAR(128) VAR, CHAR(32) VAR,
                           BIT(1) ALIGNED, CHAR(128) VAR,
                          FIXED BIN, FIXED BIN);

CALL SR$SSR (template_path, list_name, overwrite, error_path,
                    error_line, code);
```

Parameters

template_path

INPUT. The pathname of the search rules file that SR$SSR should use to set the search list.

list_name

INPUT. The name of the search list that PRIMOS should set. A search list name should be limited to 22 characters. If the search list does not exist, SR$SSR creates it. If the search list already exists, SR$SSR either overwrites its contents, or adds rules to the end of the list, depending on how you set the *overwrite* argument.

overwrite

INPUT. A flag you set to indicate whether SR$SSR should overwrite existing rules in the search list. If you set *overwrite* to '0'b, SR$SSR appends your search rules to the list without affecting existing search rules. If you set *overwrite* to '1'b, SR$SSR overwrites (deletes) existing search rules.

error_path

OUTPUT. The pathname of an unlocatable search rules file, or a search rules file containing invalid rules. If SR$SSR fails because it cannot locate an input file, it returns the pathname of that file to *error_path*. This pathname can be the search rules file, or a file requested by a –system or –insert keyword.

error_line

OUTPUT. The line number within a search list of an invalid search rule. SR$SSR returns the line number of the –insert keyword search rule that requests a circular reference. SR$SSR does not set *error_line* for –insert or –system keywords that refer to nonexistent files. The default value for this argument is 0.

code

OUTPUT. Standard error code. Possible values are

E$OK — Operation succeeded.

E$BPAR — Either the search rules file or a file invoked by an –insert or –system keyword contains invalid rules.

E$NRIT — You do not have read access rights to a file.

E$FNTF — Either the search rules file does not exist or an –insert or –system keyword refers to a nonexistent file.

E$LIST — Illegal *list_name.*

E$NEST — The –insert keyword search rules nest too deeply (over 100 levels) or request a circular reference.

Discussion

SR$SSR sets a search list by copying the rules in the search rule file specified in the *template_path* argument. It prefaces the search list with administrator rules if such rules exist for that list.

If the specified list already exists, you can direct SR$SSR to either overwrite the existing list or append rules to the existing list. An overwrite operation deletes all rules from the existing list, copies in the administrator rules for that list, then copies the rules in the *template_path* file into the search list. An append operation copies the rules in the *template_path* file to the end of the existing search list. In either event, if SR$SSR encounters an error, it sets the *code* argument and leaves the existing list unchanged.

If the search rules file you are using as a template contains an –insert keyword, SR$SSR includes the additional rules indicatedby that keyword. SR$SSR can process multiple nested inserts.

If the search rules file you are using as a template contains a –system keyword, SR$SSR inserts the system default rules at the location in your list of the –system keyword.

When performing an overwrite of an existing list, SR$SSR copies each rule's locator pointer value from the old list to the identical rule (if it exists) in the new list. This matching of search rules is case–sensitive. Refer to SR$SETL for details on locator pointers.

Examples

The following two examples perform identical operations; the first example is written in PL/I, the second in FORTRAN 77. Each of these examples sets the MYLIST search list using the MYDIR>RULES.MYLIST.SR search rules file. The *overwrite* argument instructs PRIMOS to delete all prior user–specified rules set for MYLIST.

```
/*  Sample PL/I program for the SR$SSR subroutine */

SET_SEARCH_RULES: PROCEDURE OPTIONS(MAIN);
DCL SR$SSR EXTERNAL ENTRY (CHAR(128) VAR, CHAR(32) VAR,
       BIT(1) ALIGNED, CHAR(128) VAR, FIXED BIN, FIXED BIN);
DCL FILE CHAR(128) VAR STATIC INIT(‘MYDIR>RULES.MYLIST.SR’);
DCL LIST CHAR(32) VAR STATIC INIT(‘MYLIST’);
DCL OVERWRITE  BIT(1) ALIGNED STATIC INIT(’1’b);
DCL EPATH CHAR(128) VARYING;
DCL ELINE FIXED BIN;
DCL CODE  FIXED BIN;
CALL SR$SSR(FILE, LIST, OVERWRITE, EPATH, ELINE, CODE);
IF (CODE = 0)
THEN
   PUT SKIP LIST(‘The search list has been set’);
ELSE
   BEGIN;
   PUT SKIP LIST(‘Error code:’, CODE);
   PUT SKIP LIST(‘Error path:’, EPATH);
   PUT SKIP LIST(‘Error line:’, ELINE);
   END;
PUT SKIP;
END SET_SEARCH_RULES;
```

```
C  Sample FORTRAN 77 program for the SR$SSR subroutine

C  Declarations
      INTEGER*2 FILESIZE, FILEPLUS(128)
      CHARACTER*128 FILE
      INTEGER*2 LSIZE, LPLUS(32)
      CHARACTER*32 LIST
      INTEGER*2 OVERWRITE
      INTEGER*2 EPSIZE, EPPLUS(128)
      CHARACTER*128 EPATH
      INTEGER*2 ELINE
      INTEGER*2 CODE
C  Equivalences
      EQUIVALENCE (FILESIZE, FILEPLUS(1))
      EQUIVALENCE (FILEPLUS(2), FILE)
      EQUIVALENCE (LSIZE, LPLUS(1))
      EQUIVALENCE (LPLUS(2), LIST)
      EQUIVALENCE (EPSIZE, EPPLUS(1))
      EQUIVALENCE (EPPLUS(2), EPATH)
C  Assignments
      FILE(1:21) = ‘MYDIR>RULES.MYLIST>SR’
      FILESIZE = 21
      LIST(1:6) = ‘MYLIST’
      LSIZE = 6
      OVERWRITE = :100000
C  Subroutine call
      CALL SR$SSR(FILEPLUS, LPLUS, OVERWRITE, EPATH, ELINE, CODE)
      IF (CODE.NE.0) GO TO 10
      PRINT *, ‘The search list has been set’
      CALL EXIT
C  Error processing
10    PRINT *, ‘Error code: ’, CODE
      PRINT *, ‘Error path:’, CODE
      PRINT *, ‘Error line: ’, CODE
      CALL EXIT
      END
```

Loading and Linking Information

V–mode and I–mode: No special action.

V–mode and I–mode with unshared libraries: Load NPFTNLB.

R–mode: Not available.

Obsolete File System Subroutines

A

This appendix contains descriptions of several file system subroutines that are considered obsolete and have been replaced by newer ones. The new subroutines either perform the functions of the older ones more efficiently, or have enhanced functionality, or both. In many cases the calls to the new subroutines are simpler than those of the older ones.

For programs written for use with Rev. 20.2 and later revisions, Prime encourages the use of the new subroutines in place of those described in this appendix. The older ones are presented here only for reference in maintaining programs that currently call them. When replacing these programs, you should consider using the newer calls, described elsewhere in this volume.

ATCH$$

Note In new programming, use of the AT$ subroutines in place of the ATCH$$ subroutine is recommended. The AT$ subroutines are described in Chapter 3.

ATCH$$ attaches to a directory and, optionally, makes it the home directory. In attaching to a directory, the subroutine ATCH$$ 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 directory, or in the home directory.

Usage

CALL ATCH$$ (***dirnam, namlen, ldisk, passwd, key, code***);

Parameters

dirnam

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

namlen

The length in characters (1–32) of *dirnam* (INTEGER*2). *namlen* may be greater than the length of *dirnam* provided that *dirnam* is padded with the appropriate number of blanks. If *dirnam* = K$HOME, *namlen* is disregarded.

ldisk

The number of the logical disk to be searched for *dirnam* when *key* = K$IMFD (INTEGER*2). The parameter *ldisk* must be a logical disk that is started up. Other values for *ldisk* are

K$ALLD	Search all started–up *local* logical devices in logical device order (then likewise all such *remote* devices), and attach to the directory in which *dirnam* appears in the MFD of the lowest numbered logical device.
K$CURR	Search the MFD of the disk currently attached.

passwd

A three–halfword integer array containing one of the passwords of *dirnam*. *passwd* can be specified as 0 if attaching to the home directory. If the reference subkey is K$IMFD or K$ICUR, *passwd* must be the name of a three–halfword array that specifies one of the passwords of *dirnam*. If *passwd* is blank, it must be specified as three halfwords, 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$IMFD	Attach to *dirnam* in MFD on *ldisk*.
K$ICUR	Attach to *dirnam* in current directory (*dirnam* is a subdirectory).

The SETHOME subkey, K$SETH, may be added to the REFERENCE subkey as K$IMFD+K$SETH, which will set the current directory to the home directory after attaching. If the REFERENCE subkey is K$ICUR, or if *dirnam* 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 directory (formerly referred to as the User File Directory or 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 directory is referred to as the **current directory**. As an option, this information may be copied to another place in the system, referred to as the **home directory**. The ATCH$$ subroutine does not change the home directory 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 II.

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

- Attach to home directory:

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

- Attach to directory named 'G.S.PATTON', password 'CHARGE' in current directory:

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

CREA$$

Note In new programming, use of the DIR$CR subroutine in place of the CREA$$ subroutine is recommended. This subroutine is described in Chapter 4.

CREA$$ creates a new subdirectory in the current directory and initializes the new entry. The new subdirectory is of the same type (ACL or non–ACL) as the current directory.

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*);

Parameters

filnam

The name to be given the new directory (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 directory. If *opwner_pw*(1) = 0, the owner password is set to blanks. *owner_pw* is ignored if an ACL directory is being created.

nonowner_pw

A six–character array containing the nonowner password for the new directory. 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 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

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

CREA$$ requires Add access on the current directory.

Example

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

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

CREPW$

Note In new programming, use of the DIR$CR subroutine in place of the CREPW$ subroutine is recommended. This subroutine is described in Chapter 4.

CREPW$ creates a new password directory. Add access is required on the current directory.

Usage

```
DCL CREPW$ ENTRY (CHAR(32), FIXED BIN, CHAR(6), CHAR(6),
                  FIXED BIN);
```

CALL CREPW$ (***name, name_length, owner_pw, nonowner_pw, code***);

Parameters

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.

RDEN$$

Note In new programming, use of the DIR$RD or ENT$RD subroutine in place of the RDEN$$ subroutine is preferred. These subroutines are described in Chapter 4.

RDEN$$ positions in or reads from a directory.

Usage

CALL RDEN$$ (*key, funit, buffer, buflen, rnhw, filnam, namlen, code*);

Parameters

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 directory entry and read as much of the entry as will fit into *buffer*. Set *rnhw* to the number of halfwords read.

K$NAME — Position to the start of the entry specified by *filnam* and *namlen*. Read as much of the entry as will fit into *buffer*. Set *rnhw* to the number of halfwords read. If the entry is not in the directory, the code E$FNTF is returned. If *namlen* is 0, the next entry is returned.

K$GPOS — Return the current position in the directory as a 32–bit integer in *filnam*.

K$UPOS — Set the current position in the directory 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 directory is currently opened for reading (INTEGER*2). (A directory may be opened with a call to SRCH$$.)

buffer

A one–dimensional array into which entries of the directory are read.

buflen

The length, in halfwords, of *buffer* (INTEGER*2) set to a value of 24.

rnhw

An INTEGER*2 variable that will be set to the number of halfwords 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. Possible values are

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

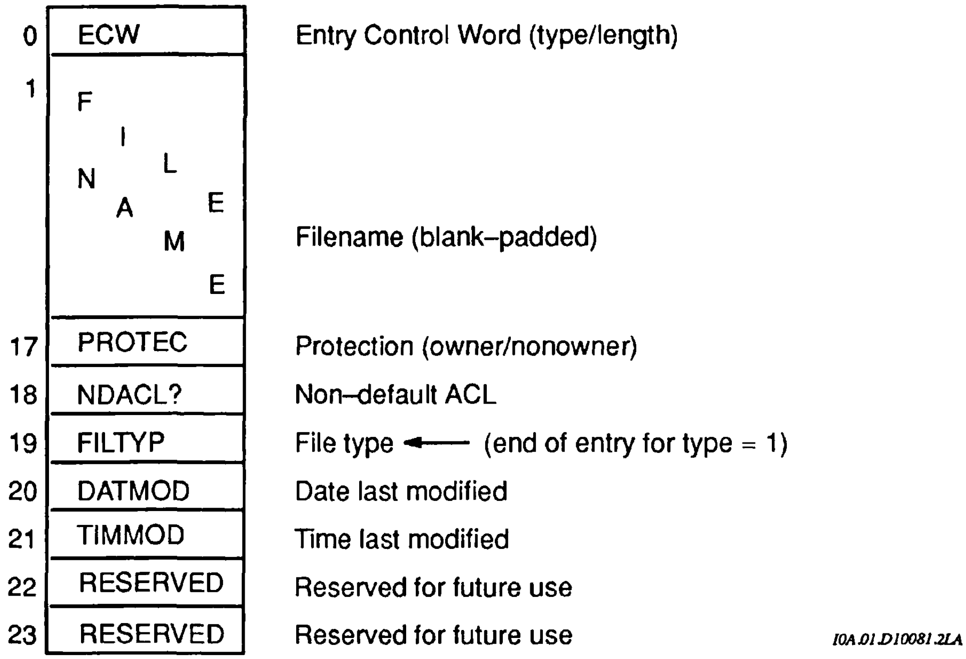
Discussion

RDEN$$ is used to read entries from a directory. *rnhw* halfwords 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, directories 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 directory.

The complete format of currently defined entries is given in Figure A–1 and discussed below for revisions before Rev. 19. (For Rev. 19 format, see DIR$RD.) All numbers are decimal unless preceded by a colon (:).



0	ECW	Entry Control Word (type/length)
1	FILENAME	Filename (blank–padded)
17	PROTEC	Protection (owner/nonowner)
18	NDACL?	Non–default ACL
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

Figure A–1. File Entry Format

ECW

Entry Control Word. An ECW is the first halfword 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 halfwords including the ECW itself. Possible values of the ECW are as follows:

:003030	Type=3, length=24. A type of 3 indicates an access category directory entry. All the above information is 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

non_default_ acl

The high–order bit is 1 if this UFD entry is protected by a specific ACL or access category, 0 if it is protected by the default ACL. Bits 2–16 are reserved.

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
6	Access category

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 II) 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 systemwide basis.

The meaning of the lock values is

Value	*Bits 5, 6*	*Meaning*
0	0, 0	Use systemwide 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.

Examples

- 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
```

- Position to beginning of UFD:

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

- 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(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 */
/***************************************************************/
```

- 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(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;
```

TSRC$$

Note In new programming, use of the SRSFX$ subroutine in place of the TSRC$$ subroutine is recommended. This subroutine is described in Chapter 4.

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

Usage

CALL TSRC$$ (*action+newfil, pathname, funit, chrpos, type, code*);

Parameters

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$GETU	Open *pathname* on an unused file unit selected by PRIMOS. The unit number is returned in *funit.*
K$VMR	Open *pathname* for VMFA read.

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 halfword.

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 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.

B Data Type Equivalents

To call a subroutine from a program written in any Prime language, you must declare the subroutine and its parameters in the calling program. Therefore, you must translate the PL/I data types expected by the subroutine into the equivalent data types in the language of the calling program.

Table B–1 shows the equivalent data types for the Prime languages BASIC/VM, C, COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, and PL/I. The leftmost column lists the generic storage unit, which is measured in bits, bytes, or halfwords for each data type. Each storage unit matches the data types listed to the right on the same row. The table does not include an equivalent data type for each generic unit in all languages. However, with knowledge of the corresponding machine representation, you can often determine a suitable workaround. For instance, to see if you can use a left–aligned bit in COBOL 74, you could write a program to test the sign of the 16–bit field declared as COMP. In addition, if a subroutine parameter consists of a structure with elements declared as BIT(*n*), it can be declared as an integer in the calling program. Read the appropriate language chapter in the *Subroutines Reference I: Using Subroutines* before using any of the equivalents shown in the table.

Note The term PL/I refers both to full PL/I and to PL/I Subset G (PL/I–G).

Table B-1. Equivalent Data Types for Prime Languages

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
16-bit integer	INT	short enum	COMP PIC S9(1)- PIC S9(4)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER Enumerated	FIXED BIN FIXED BIN(15)
32-bit integer	INT*4	int long	COMP PIC S9(5)- PIC S9(9)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	FIXED BIN(31)
64-bit integer			COMP PIC S9(10)- PIC S9(18)				
32-bit float single precision	REAL	float	COMP-1	REAL REAL*4	REAL REAL*4	REAL	FLOAT BIN FLOAT BIN(23)
64-bit float double precision	REAL*8	double	COMP-2	REAL*8	REAL*8	LONGREAL	FLOAT BIN(47)
128-bit float quad precision					REAL*16		
1 bit		short					BIT BIT(1)
1 left-aligned bit		short				BOOLEAN	BIT(1) ALIGNED

T0F01.D10081.2LA

Table B-1. Equivalent Data Types for Prime Languages (Continued)

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
Bit string		unsigned int				SET	BIT(n)
Fixed-length character string	INT	char NAME[n] char NAME	DISPLAY PIC A(n) PIC X(n) FILLER		CHARACTER *n	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)
Fixed-length digit string			DISPLAY PIC 9(n)				PICTURE
Fixed-length digit string, 2 digits per byte			COMP-3				FIXED DECIMAL
Varying-length character string		struct {short LENGTH; char DATA[n]; } CVAR				STRING[n]	CHAR(n) VARYING
32-bit pointer		Pointer (32IX-mode)		LOC( )	LOC( )		POINTER OPTIONS (SHORT)
48-bit pointer		Pointer (64V-mode)				Pointer	POINTER

T0F.01.D10081.2LA

Notes

For a discussion of possible workarounds for some of the empty boxes in this table as well as a description of generic units for PMA, refer to the appropriate language chapter in the *Subroutines Reference I: Using Subroutines.*

The BASIC/VM column lists FTN data types to be declared in the SUB FORTRAN statement in a BASIC/VM program.

Argument Parsing by the CL$PIX Subroutine

C

Overview

The CL$PIX subroutine allows a program to process arguments on a command line, using the rules explained for arguments in the *CPL User's Guide.*

Using a description of the expected arguments in the form of a list of keywords, CL$PIX builds a structure consisting of a number of elements, or pixels, containing the arguments in a readily accessible form for the routine that is to use the arguments.

CL$PIX Operating Modes

CL$PIX operates in either of two modes: a **normal** mode for routines that call for and use arguments entirely within themselves, and **CPL** mode for routines that are called by CPL programs and pass the parsed arguments back to the calling program. The two modes differ principally in the way in which they point to the parsed argument structure. They are described in detail on the following pages.

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. Uppercase 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.

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) through (–2**31 + 1).

hex

Hexadecimal integer, unsigned, in the range 0 through (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.

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.) Uppercase 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. May be as long as 160 bytes. Uppercase and lowercase are distinguished. See the discussion of data type UNCL below.

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

Data Type	*PL/I Type*	*FORTRAN Type*
char	char(80) var	INTEGER(41)
char1	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)

The following examples show the declaration statements for specific data types in a picture.

Picture	*Structure*
char; end	dcl 1 struc, 2 char_arg char(80) var;
tree; dec; char1; end	dcl 1 struc, 2 tree_arg char(128) var, 2 dec_arg fixed bin(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 be used only 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, if the picture is

```
dec; tree; rest; end
```

and the structure is

```
dcl 1 struc,
     2 dec_arg fixed bin(31),
     2 tree_arg char(128) var,
     2 rest_arg char(160) var;
```

then, for the command line

```
786 a>b>c>d foo 99 zot>nil
```

the value 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 a default value specified as described below, 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 bin(31),
         2 date_arg fixed bin(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 bin(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 bin(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

```
-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 objects in the general form, and are specified using the syntax for object argument groups, except that no semicolon is used between objects. 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 bin(31),
      2 range_upper fixed bin(31);

-range 7        (command line)
```

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 bin(31);
```

When the *–string* option 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 bin(31);
```

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; for example, 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,
          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 the same as that accepted by the CPL &ARGS directive. (In fact, CPL uses CL$PIX in CPL mode to process the &ARGS directive.) The *CPL User's Guide* gives details on the syntax and parsing of the &ARGS directive.

The salient differences between CL$PIX syntaxes in normal mode and CPL mode are

- Repeat counts are not allowed in CPL mode.
- Each object argument and option argument must be preceded by a variable identifier terminated with a colon, thus:

  ```
  path:tree; time_of_day:-time dec; unclaimed:UNCL
  ```

  where *path*, *time_of_day*, and *unclaimed* are CPL local variable names. The value of each argument is assigned to the local variable whose name is prefixed to that argument.
- The *end* token is not used in CPL mode, and a semicolon is not required after the last token.
- The maximum length of any argument value in CPL mode is 1024 characters, unlike normal mode where the limit depends on the data type (80 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.

Example for CL$PIX

The following example uses CL$PIX to parse a command line.

```
test:
        proc;

/* EXTERNAL ENTRY POINTS */

dcl cl$get entry (char(*)var, fixed bin, fixed bin),
    cl$pix entry (bit(16) aligned, char(*)var, ptr,
                  fixed bin,char(*)var, ptr, fixed bin,
                  fixed bin, fixed bin, ptr),
    errpr$ entry (fixed bin, fixed bin, char(*), fixed
                  bin, char(*), fixed bin,
    tnoua entry (char(*), fixed bin),
    todec entry (fixed bin),
    tnou entry (char(*), fixed bin);

/* INSERT FILES */

$Insert syscom>keys.ins.pl1

/* LOCAL DECLARATIONS */

dcl code fixed bin,             /* standard error code */
    non_st_code fixed bin,      /* cl$pix error code   */
    pix_index fixed bin,
    bad_index fixed bin,
    picture char(30) var,
    pic_ptr ptr,
    out_ptr ptr,
    arg_line char(150) var;
```

```
dcl 1 args,
       2 dir char(128) var,
       2 file char(32) var;

dcl 1 bvs based,
       2 len fixed bin,
       2 chars char(1);

/* PROMPT USER FOR ARGUMENTS */

call tnoua('Enter directory pathname and filename: ', 38);

/* READ IN ARGS TO CALL */

call cl$get (arg_line, 150, code);
if code ^= 0
   then call er$print(k$nrtn, ssc$errd, code,
                     'CANNOT READ ARGS', 16, 'test', 9);
else do;

/* SET UP DATA FOR CL$PIX */

   picture = 'tree; entry; end';
   pic_ptr = addr(picture);
   out_ptr = addr(args);

/* CALL CL$PIX TO PARSE ARGUMENTS */

   call cl$pix('0003'b4, 'test', pic_ptr, 30, arg_line,
               out_ptr, pix_index, bad_index, non_st_code,
                null());
   if non_st_code ^= 0
      then do;
      call tnoua('CANNOT PARSE ARGS, error code = ', 32);
      call todec(non_st_code);
      call tnou(' ', 1);
      end;

/* OUTPUT ARGUMENTS READ IN */

   else do;
      call tnoua('Directory pathname = ', 21);
      call tnou(addr(dir) -> bvs.chars, addr(dir) ->
              bvs.len);
      call tnoua('File name = ', 12);
   call tnou(addr(file) -> bvs.chars, addr(file) ->
           bvs.len);
      end;
   end;
end;
```

The previous program gives the following output:

```
Enter directory pathname and filename:
<testpk>my_ufd my_file
Directory pathname = <TESTPK>MY_UFD
File name = MY_FILE
```

Calls Made by CL$PIX

TNCHK$
FNCHK$
IDCHK$
PWCHK$

Index of Subroutines by Function

This index lists subroutines grouped by the general functions that they perform. See the Index of Subroutines by Name to find a particular subroutine's volume, chapter, and page number.

Access Category

Add an object's name to an access category.	AC$CAT
Modify an existing ACL on an object.	AC$CHG
Set an object's ACL to that of its parent directory.	AC$DFT
Make an object's ACL identical to that of another object.	AC$LIK
Obtain the contents of an object's ACL.	AC$LST
Convert an object from ACL protection to password protection.	AC$RVT
Set a specific ACL on an object.	AC$SET
Determine whether an object is accessible for a given action.	CALAC$
Delete an access category.	CAT$DL
Obtain the user-ID and the groups to which it belongs.	GETID$
Obtain the passwords of a subdirectory of the current directory.	GPAS$$
Determine whether an object is ACL-protected.	ISACL$
Remove an object's priority access.	PA$DEL
Obtain the contents of an object's priority ACL.	PA$LST
Set priority access on an object.	PA$SET
Set the owner and nonowner passwords on an object.	SPAS$$

Access Server Names

Catalog a server's Low Level Name.	ISN$C
Look up a server's Low Level Name.	ISN$L
Recatalog a server's Low Level Name.	ISN$RC
Uncatalog a server's Low Level Name.	ISN$UC
Get the server name of a process.	SRS$GN

Get the process numbers of all processes associated with the server name.	SRS$GP
List the server names on your system.	SRS$LN

Arrays

Get a character from an array.	GCHAR
Store a character into an array location.	SCHAR

Asynchronous Lines

Return asynchronous line characteristics.	AS$LST
Return an asynchronous line number.	AS$LIN
Set asynchronous line characteristics.	AS$SET

Attach Points

Set the attach point to a directory specified by the pathname.	AT$
Set the attach point to a specified top-level directory and partition.	AT$ABS
Set the attach point to a specified top-level directory on any partition.	AT$ANY
Set the attach point to the home directory.	AT$HOM
Set the attach point to a specified top-level directory on a partition identified by logical disk number.	AT$LDEV
Set the attach point to the login directory.	AT$OR

Set the attach point to a directory subordinate to the current directory.	AT$REL
Set the attach point to the root directory.	AT$ROOT
Set the attach point to a specified directory, and optionally, make it the home directory.	ATCH$$

Binary Search

Perform binary search in ordered table.	BIN$SR

Buffer Output

Provide free-format output to a buffer.	IOA$RS

Command Environment

Return caller's maximum command environment breadth.	CE$BRD
Return caller's maximum command environment depth.	CE$DPT
Parse command arguments according to a character string "picture" of the command line.	CL$PIX
Invoke a command from a running program.	CP$
Retrieve the value of a global variable.	GV$GET
Set the value of a global variable.	GV$SET
Return a list of commands valid at mini-command level.	LIST$CMD
Retrieve the value of a CPL local variable.	LV$GET

Set the value of a CPL local variable.	LV$SET
Return breadth of caller's current command environment.	RD$CE_DP

Command Level

Call a new command level after an error.	CMLV$E
Call a new command level.	COMLV$
Return to PRIMOS.	EXIT
Initialize the command environment.	ICE$
Return serialization data.	KLM$IF
Record command error status.	SETRC$
Signal an error in a subsystem.	SS$ERR

Condition Mechanism

Continue scan for on-units.	CNSIG$
Convert FORTRAN statement label to PL/I format.	MKLB$F
Create an on-unit (for FTN users).	MKON$F
Create an on-unit (for any language except FTN).	MKON$P
Create an on-unit (for PMA and PL/I users).	MKONU$
Perform a nonlocal GOTO.	PL1$NL
Revert an on-unit (for FTN users).	RVON$F
Revert an on-unit (for any language except FTN).	RVONU$
Signal a condition (for FTN users).	SGNL$F
Signal a condition (for any language except FTN.)	SIGNL$

Controllers, Asynchronous, Multi-line

Communicate with SMLC driver.	T$SLC0
Assign AMLC line.	ASNLN$
Communicate with AMLC driver.	T$AMLC

Data Conversion

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$A
Convert the TIMMOD field (as returned by RDEN$A).	FTIM$A

Date Formats

Convert binary date to quadseconds.	CV$DQS
Convert ASCII date to binary format.	CV$DTB
Convert binary date to ISO format.	CV$FDA
Convert binary date to visual format.	CV$FDV
Convert quadsecond date to binary format.	CV$QSD

Devices, Assigning or Attaching

Attach specified devices.	ATTDEV
Provide or set aside available logical file unit.	IOCS$G
Free a logical file unit number.	IOCS$F

Disk I/O

Read ASCII from disk.	I$AD07
Write binary to disk.	O$BD07
Read binary from disk.	I$BD07
Write ASCII to disk (fixed-length records).	O$AD08
Register disk format with driver.	DKGEO$

Drivers, Device-independent

Write ASCII data.	WRASC
Read ASCII data.	RDASC
Write binary data.	WRBIN
Read binary data.	RDBIN
Open PRIMOS file and perform other nondata transfer functions. (Primarily for IOCS applications.)	CONTRL

Encryption, of Login Password

Encrypt login validation passwords.	ENCRYPT$

EPFs

Allocating and Deallocating Space for EPFs

Allocate space for EPF function return information.	ALC$RA
Allocate space and set value of EPF function return information.	ALS$RA
Deallocate space for EPF function return information.	FRE$RA

Management of EPFs

Perform the linkage allocation phase for an EPF.	EPF$ALLC
Return the state of the command processing flags in an EPF.	EPF$CPF
Deactivate the most recent invocation of a specified EPF.	EPF$DEL
Perform the linkage initialization phase for an EPF.	EPF$INIT
Initiate the execution of a program EPF.	EPF$INVK
Map the procedure images of an EPF file into virtual memory.	EPF$MAP
Combine functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.	EPF$RUN
Modify user's search rules to allow dynamic linking to a library EPF.	LN$SET
Remove an EPF from a user's address space.	REMEPF$
Replace one EPF runfile with another.	RPL$

Information From In-memory User Profile

Return maximum number of dynamic segments.	DY$SGS
Return maximum number of static segments.	ST$SGS
Return highest segment number.	TL$SGS

Registering EPFs

Return ready or suspended status for registered EPF.	EPF$ISREADY
Enable registration of EPFs	EPF$REG
Enable unregistration of registered EPFs	EPF$UREG

Error Handling, I/O

Set ERRVEC and perform a return or display ERRVEC message before returning control to system.	ERRSET
Obtain contents of ERRVEC.	GETERR
Display I/O error message on user terminal.	PRERR

Event Synchronizers and Event Groups

Creating, Using, and Destroying Event Synchronizers

Create an event synchronizer.	SYN$CREA
Post a notice on an event synchronizer.	SYN$POST
Wait on an event synchronizer.	SYN$WAIT
Perform a timed wait on an event synchronizer.	SYN$TMWT

Retrieve a notice from an event synchronizer.	SYN$RTRV
Destroy an event synchronizer.	SYN$DEST

Creating, Using, and Destroying Event Groups

Create an event group.	SYN$GCRE
Move an event synchronizer into an event group.	SYN$MVTO
Remove an event synchronizer from an event group.	SYN$REMV
Cause a process to wait on an event group.	SYN$GWT
Cause a process to perform a timed wait on an event group.	SYN$GTWT
Retrieve a notice from an event group.	SYN$GRTR
Destroy an event group.	SYN$GDST

Getting Information About Synchronizers and Groups

Return number of notices or waiting processes on a synchronizer.	SYN$CHCK
Return number of notices on a group at one or all priority levels; if all levels, also return number of waiting processes.	SYN$GCHK
Indicate whether synchronizer is in group; and if it is, return the group number, priority level, and For Client Use field.	SYN$INFO
List the synchronizers in group and total number.	SYN$LSIG
List the synchronizers in server and total number.	SYN$LIST
List the groups in server and total number.	SYN$GLST

Executable Images

Restore an R-mode executable image.	REST$$
Restore and resume an R-mode executable image.	RESU$$
Save an R-mode executable image.	SAVE$$

EXIT$ Condition

Disable signalling of EXIT$ condition.	EX$CLR
Return state of EXIT$ signalling.	EX$RD
Enable signalling of EXIT$ condition.	EX$SET

File System Objects

Append a specified suffix to a pathname.	APSFX$
Extend or truncate a CAM file.	CF$EXT
Retrieve a CAM file's extent map from disk.	CF$REM
Set a CAM file's allocation size value.	CF$SME
Change the open mode of an open file.	CH$MOD
Close a file by name and return a bit string indicating closed units.	CL$FNR
Close a file system object by pathname.	CLO$FN
Close a file system object by file unit number.	CLO$FU
Close a file.	CLOS$A
Change the name of an object in the current directory.	CNAM$$
Create a new subdirectory in the current directory.	CREA$$
Create a new password directory.	CREPW$

Delete a file.	DELE$A
Create a new directory.	DIR$CR
Search for specified types of entries in a directory open on a file unit.	DIR$LS
Read sequentially the entries of a directory open on a file unit.	DIR$RD
Return entries meeting caller-specified selection criteria in a directory open on a file unit.	DIR$SE
Return the contents of a named entry in a directory open on a file unit.	ENT$RD
Generate a filename based on another name.	EQUAL$
Check for file existence.	EXST$A
Return a file system object's entryname and parent directory pathname.	EXTR$A
Delete a file identified by a pathname.	FIL$DL
Return information about a specified file unit.	FINFO$
Verify a supplied string as a valid filename.	FNCHK$
Force PRIMOS to write modified records to disk.	FORCEW
Position to end-of-file.	GEND$A
Return the pathname of a specified unit, attach point, or segment.	GPATH$
Tell whether the partition on which a file exists is robust.	GTROB$
Determine whether an open file system object is local or remote.	ISREM$
Return information on the system's list of logical disks.	LDISK$
List the disks a given user is using.	LUDSK$
Convert an existing directory entry to a portal by mounting the defined portal over the directory.	NAM$AD_PORTAL
Read the contents of the Global Mount Table; return a list of current-mounted disk partitions and the currently mounted portals accessible by the calling program	NAM$L_GMT
Remove a portal entry from the specified directory pathname.	NAM$RM_PORTAL

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
Return a logical value indicating whether a specified partition supports ACL protection and quotas.	PAR$RV
Position file.	POSN$A
Read, write, position, or truncate a file.	PRWF$$
Return directory quota and disk record usage information.	Q$READ
Set a quota on a subdirectory in the current directory.	Q$SET
Position in or read from a directory.	RDEN$$
Read a line of characters from an ASCII disk file.	RDLIN$
Return position of file.	RPOS$A
Rewind file.	RWND$A
Set or modify an object's attributes in its directory entry.	SATR$$
Delete a segment directory entry.	SGD$DL
Determine if a segment directory entry exists.	SGD$EX
Open a segment directory entry.	SGD$OP
Position in, read an entry in, or modify the size of a segment directory.	SGDR$$
Return the size of a file system entry.	SIZE$
Open, close, delete, change access, or verify the existence of an object.	SRCH$$
Search for a file with a list of possible suffixes.	SRSFX$
Open a scratch file with unique name.	TEMP$A
Verify a supplied string as a valid pathname.	TNCHK$
Truncate file.	TRNC$A
Scan the file system structure.	TSCN$A
Open a file anywhere in the PRIMOS file structure.	TSRC$$
Check for file open.	UNIT$A

Return the minimum and maximum file unit numbers currently in use by this user.	UNITS$
Return a logical value indicating whether a wildcard name was matched.	WILD$
Write a line of characters to a file in compressed ASCII format.	WTLIN$

ISC

Establish an ISC Session

Initiator requests the session.	IS$RS
Recipient gets the session request.	IS$GRQ
Recipient accepts the session.	IS$AS
Initiator gets the session request response.	IS$GRS

ISC Message Exchange

Allocate a buffer for a message data part.	IS$AB
Free an allocated data part buffer.	IS$FB
Send a message.	IS$SM
Receive a message.	IS$RM

Monitor ISC Message Exchange Session

Get sessions owned by your server.	IS$GSO
Get session attributes.	IS$GSA
Get session status.	IS$GSS
Get statistics about a session.	IS$STA

Terminate ISC Sessions or Respond to Exceptions

Terminate the caller's side of a session.	IS$TS
Get an exception.	IS$GE
Clear an exception.	IS$CE

Keyboard or ASR Reader

Input ASCII from terminal or ASR reader.	I$AA01
Perform same function as I$AA01 but also allow input from a cominput file.	I$AA12

Logging

Log a user message to the DMS server.	DS$SEND_CUSTOMER_UM

Matrix Operations

Generate permutations.	PERM
Generate combinations.	COMB

The following groups contain subroutines for single-precision, double-precision, integer, and complex operations, respectively.

(* indicates that a subroutine is not available.)

Set matrix to identity matrix.	MIDN, DMIDN, IMIDN, CMIDN
Set matrix to constant matrix.	MCON, DMCON, IMCON, CMCON
Multiply matrix by a scalar.	MSCL, DMSCL, IMSCL, CMSCL
Perform matrix addition.	MADD, DMADD, IMADD, CMADD
Perform matrix subtraction.	MSUB, DMSUB, IMSUB, CMSUB
Perform matrix multiplication.	MMLT, DMMLT, IMMLT, CMMLT
Calculate transpose matrix.	MTRN, DMTRN, IMTRN, CMTRN
Calculate adjoint matrix.	MADJ, DMADJ, IMADJ, CMADJ
Calculate inverted matrix.	MINV, DMINV, *, CMINV
Calculate signed cofactor.	MCOF, DMCOF, IMCOF, CMCOF
Calculate determinant.	MDET, DMDET, IMDET, CMDET
Solve a system of linear equations.	LINEQ, DLINEQ, *, CLINEQ

Memory

Allocate memory on the current stack.	ALOC$S
Move a block of memory.	MOVEW$
Make the last page of a segment available.	MM$MLP
Make the last page of a segment unavailable.	MM$MLP
Allocate user-class dynamic memory.	STR$AL
Allocate process-class dynamic memory.	STR$AP
Allocate subsystem-class dynamic memory.	STR$AS

Allocate user-class dynamic memory.	STR$AU
Free process-class dynamic memory.	STR$FP
Free user-class dynamic memory.	STR$FR
Free subsystem-class dynamic memory.	STR$FS
Free user-class dynamic memory.	STR$FU

Message Facility

Return the receiving state of a user.	MSG$ST
Set the receiving state for messages.	MGSET$
Receive a deferred message.	RMSGD$
Send an interuser message.	SMSG$

Numeric Conversions

Convert string (decimal) to 16-bit integer.	CH$FX1
Convert string (decimal) to 32-bit integer.	CH$FX2
Convert string (hexadecimal) to 32-bit integer.	CH$HX2
Convert string (octal) to 32-bit integer.	CH$OC2

Paper Tape

Control functions for paper tape.	C$P02
Input ASCII from the high-speed paper-tape reader.	I$AP02
Output binary data to the high-speed paper-tape punch.	O$BP02

Input one character from the high-speed paper-tape reader to Register A.	P1IB
Output one character to the high-speed paper-tape punch from Register A.	P1OB
Input one character from paper tape, set high-order bit, ignore line feeds, send a line feed when carriage return is read.	P1IN
Output one character to the high-speed paper-tape punch.	P1OU

Parsing

Parse a PRIMOS command line.	CMDL$A
Parse character string into tokens.	GT$PAR

Peripheral Devices

Line Printers

Centronics LP.	O$AL04
Parallel interface to line printer (MPC).	O$AL06
Versatec printer.	O$AL14
Move data to MPC line printer.	T$LMPC
Access a spooler queue.	SPOOL$
Place file in spool queue and perform SPOOLER command functions.	SP$REQ

Printer/Plotter

Versatec.	O$AL14
Versatec.	T$VG

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

Write EBCDIC to 9-track.	O$AM13
Read EBCDIC from 9-track.	I$AM13
Raw data mover.	T$MT

Phantom Processes

Switch logout notification on or off.	LON$CN
Read logout notification information.	LON$R
Start a phantom process.	PHNTM$

Process Suspension

Suspend a process for a specified interval.	SLEEP$
Suspend a process (interruptible).	SLEP$I

Query User

Prompt and read a name.	RNAM$A
Prompt and read a number (binary, decimal, octal, or hexadecimal).	RNUM$A
Ask question and obtain a YES or NO answer.	YSNO$A

Randomizing

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

Search Rules

Locate a file using a search list and open the file. Create a file if the file sought does not exist.	OPSR$
Locate a file using a search list and a list of suffixes. Open the located file, or create a file if the file sought does not exist.	OPSRS$
Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL.	SR$ABSDS

Add a rule to the beginning of a search list or before a specified rule.	SR$ADDB
Add a rule to the end of a search list or after a specified rule.	SR$ADDE
Create a search list.	SR$CREAT
Delete a search list.	SR$DEL
Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL.	SR$DSABL
Enable an optional search rule. Enabled rules can be disabled using SR$DSABL or SR$ABSDS.	SR$ENABL
Determine if a search rule exists.	SR$EXSTR
Free list structure space allocated by SR$LIST or SR$READ.	SR$FR_LS
Initialize all search lists to system defaults.	SR$INIT
Return the names of all defined search lists.	SR$LIST
Read the next rule from a search list.	SR$NEXTR
Read all of the rules in a search list	SR$READ
Remove a search rule from a search list.	SR$REM
Set the locator pointer for a search rule.	SR$SETL
Set a search list using a user-defined search rules file.	SR$SSR

Semaphores

Release (close) a named semaphore.	SEM$CL
Drain a semaphore.	SEM$DR
Notify a semaphore.	SEM$NF
Open a set of named semaphores.	SEM$OP
Open a set of named semaphores.	SEM$OU
Periodically notify a semaphore.	SEM$TN
Return number of processes waiting on a semaphore.	SEM$TS

Wait on a specified named semaphore, with timeout.	SEM$TW
Wait on a semaphore.	SEM$WT

Sorting

Sort one file on ASCII key(s).	SUBSRT
Sort (multiple key types) or merge sorted files.	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
Sort tables prepared by SETU$S.	CMBN$S
Get sorted records.	RTRN$S
Close all sort units.	CLNU$S
Heap sort.	HEAP
Partition exchange sort.	QUICK
Diminishing increment sort.	SHELL
Radix exchange sort.	RADXEX
Insertion sort.	INSERT
Bubble sort.	BUBBLE
Binary search or build binary table.	BNSRCH

Strings

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$A
Compare two character strings.	NAMEQ$
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$A
Return unique bit string.	UID$BT
Convert UID$BT output into character string.	UID$CH

System Administration

General System Administration

Change the user ID of the System Administrator.	CUS$CHANGE_ADMIN
Enable changes to the system attributes.	CUS$CHANGE_SYSTEM
Check System Administration Directory (SAD) hashing status for the system or a project.	CUS$CHECK_SAD
Close a SAD that has been opened.	CUS$CLOSE_SAD
Create a System Administration Directory (SAD).	CUS$CREATE_SAD
List the attributes of the overall system.	CUS$LIST_SYSTEM
Open an existing System Administration Directory (SAD).	CUS$OPEN_SAD
Rebuild the SAD for either the system or a project.	CUS$REBUILD_SAD
Check if the user is the System Administrator of the open SAD.	CUS$SA_MODE

Group Administration

Check if an ACL group is already a system ACL group or project ACL group.	CUS$CHECK_GROUP
Add an ACL group to the SAD.	CUS$GROUP
List the system and project ACL groups.	CUS$LIST_GROUP_NAMES
List the projects using an ACL group.	CUS$LIST_GROUPS_PROJECTS
List the users of a system or project ACL group.	CUS$LIST_GROUPS_USERS

Project Administration

Check if a project is on the system.	CUS$CHECK_PROJECT_ID
List the projects using an ACL group.	CUS$LIST_GROUPS_PROJECTS

List the attributes of a specific project.	CUS$LIST_PROJECT
Add, delete, or change a specific project.	CUS$PROJECT

User Administration

Check if a user is on the system or a member of a project.	CUS$CHECK_USER_ID
List the users of a system or project ACL group.	CUS$LIST_GROUPS_USERS
List the attributes of a specific user.	CUS$LIST_USER
List the users on the system or on a project.	CUS$LIST_USER_NAMES
Add, delete, or change a specific user.	CUS$USER
Check the network to see if a particular user ID is valid on other machines.	CUS$VERIFY_USER

System Information

General System Information

Return cold-start setting of the ABBREV switch.	AB$SW$
Determine if the routine is dynamically accessible.	CKDYN$
Return text of the specified system prompt.	CL$MSG
Return the model number of the Prime computer.	CPUID$
Return the current date and time.	DATE$
Return text representation of an error code.	ERTXT$
Return text representation of an error code for specified PRIMOS subsystem.	ER$TEXT
Return PRIMOS II information.	GINFO
Return the current PRIMOS system name.	GSNAM$
Return information on the system's list of logical disks.	LDISK$
Indicate if login-over-login is permitted.	LOV$SW

Return information about a PRIMOS line used for LAN terminal service.	NT$LTS
Return the operating system revision number.	PRI$RV
Determine access to a segment.	RSEGAC$
Check validity of a system name passed to it.	SNCHK$
Return the user number and count of users.	USER$

System Time Information

Return the CPU time since login.	CTIM$A
Return today's date, American style.	DATE$A
Return today's date as day of year (the Julian date).	DOFY$A
Return the disk time since login.	DTIM$A
Return today's date, European (military) style.	EDAT$A
Return the time of day.	TIME$A

System Status and Metering Information

Return data about a disk partition.	DS$AVL
Return data about a process's environment.	DS$ENV
Return data about file units.	DS$UNI
Return a variety of metering information.	G$METR

Timers

Set and read various timers.	LIMIT$
Create a timer.	TMR$CREA
Destroy a timer.	TMR$DEST

Set an absolute timer.	TMR$SABS
Set an interval timer.	TMR$SINT
Set a repetitive timer.	TMR$SREP
Cancel a timer.	TMR$CANL
Return the timer type and information.	TMR$GTMR
List the identifiers of the timers within a server.	TMR$LIST

User Information

Check that a process has a given amount of time slice left.	ASSUR$
Change login validation password.	CHG$PW
Expand a line using abbreviations preprocessor.	COM$AB
Generate a new login validation password.	GEN$PW
Validate a name.	IDCHK$
Determine whether a forced logout is in progress.	IN$LO
List the disks a given user is using.	LUDSK$
Log out a user.	LOGO$$
Return a list of devices that a user can access.	LUDEV$
Return the user's project identifier.	PRJID$
Return amount of CPU time used since login.	PTIME$
Validate syntax of a password.	PWCHK$
Display PRIMOS command prompt.	READY$
Return user number of initiating process.	SID$GT
Test whether current user is supervisor.	SUSR$
Display standard message showing times used.	TI$MSG
Return timing information and user identification.	TIMDAT
Return permanent time information.	TMR$GINF
Return current system time.	TMR$GTIM

Convert local time to Universal Time.	TMR$LOCALCONVERT
Convert Universal Time to local time.	TMR$UNIVCONVERT
List users with same name as caller.	UNO$GT
Return user type of current process.	UTYPE$
Validate a name against composite identification.	VALID$

User Terminal

Functions

Control functions for user terminal.	C$A01
Output ASCII to the user terminal or ASR punch.	O$AA01
Inhibit or enable CONTROL–P.	BREAK$
Get next character from terminal or command file.	C1IN
Get next character from command line until carriage return.	C1IN$
Move characters from terminal or command file to memory.	CNIN$
Read a line of text from the terminal or from a command file.	COMANL
Supervise the editing of input from a terminal or a command file (callable from C).	ECL$CC
Supervise the editing of input from a terminal or a command file.	ECL$CL
Read or set erase and kill characters.	ERKL$$
Output *count* characters to the user terminal followed by a line feed and carriage return.	TNOU
Output *count* characters to the user terminal.	TOVFD$
Read one character from the user terminal into Register A.	T1IB
Read one character from the user terminal.	T1IN

Write one character from Register A to the user terminal.	T1OB
Output *char* to the user terminal. The data type must be a 16-bit integer in F77.	T1OU
Input decimal number.	TIDEC
Input an octal number.	TIOCT
Input a hexadecimal number.	TIHEX
Output a six-character signed decimal number.	TODEC
Output a six-character unsigned octal number.	TOOCT
Output a four-character unsigned hexadecimal number.	TOHEX
Output carriage return and line feed.	TONL

Input From User Terminal

Read a character.	C1IN
Read a character.	C1IN$
Read a character, suppressing echo.	C1NE$
Read a line.	CL$GET
Read a specified number of characters.	CNIN$
Read a line into a PRIMOS buffer.	COMANL
Parse a command line.	RDTK$$
Read a character (function).	T1IB
Read a character (procedure).	T1IN
Read a decimal number.	TIDEC
Read a hexadecimal number.	TIHEX
Read an octal number.	TIOCT
Check for presence of characters in user's terminal output buffer.	TTY$OUT

Output to User Terminal

Print a standard error message from PRIMOS or a PRIMOS subsystem.	ER$PRINT
Print a standard error message.	ERRPR$
Provide free-format output.	IOA$
Provide free-format output, for error messages.	IOA$ER
Write characters to terminal, followed by NEWLINE.	TNOU
Write characters to terminal.	TNOUA
Write a signed decimal number.	TODEC
Write a hexadecimal number.	TOHEX
Write a NEWLINE.	TONL
Write an octal number.	TOOCT
Write a decimal number, without spaces.	TOVFD$
Write one character from Register A.	T1OB
Write one character.	T1OU

Control Output to User Terminal

Inhibit or enable BREAK function.	BREAK$
Return information about command output settings.	CO$GET
Switch input between the terminal and a file.	COMI$$
Switch output between the terminal and a file.	COMO$$
Control the way PRIMOS treats the user terminal.	DUPLX$
Read or set the erase and kill characters.	ERKL$$
Determine if there are pending quits.	QUIT$
Check for unread terminal input characters.	TTY$IN
Clear the terminal input and output buffers.	TTY$RS

Index of Subroutines by Name

A$xy series	FORTRAN compiler addition functions.	I	B–7
AB$SW$	Return cold-start setting of ABBREV switch.	III	2–3
AC$CAT	Add an object's name to an access category.	II	2–3
AC$CHG	Modify an existing ACL on an object.	II	2–5
AC$DFT	Set an object's ACL to that of its parent directory.	II	2–7
AC$LIK	Set an object's ACL like that of another object.	II	2–9
AC$LST	Obtain the contents of an object's ACL.	II	2–11
AC$RVT	Convert an object from ACL protection to password protection.	II	2–13
AC$SET	Set a specific ACL on an object.	II	2–14
ALC$RA	Allocate space for EPF function return information.	III	4–16
ALOC$S	Allocate memory on the current stack.	III	4–3
ALS$RA	Allocate space and set value of EPF function.	III	4–21
APSFX$	Append a specified suffix to a pathname.	II	4–4
ASCS$$	Sort or merge sorted files (multiple file types and key types). (V-mode)	IV	17–12
ASCS$$	Sort or merge sorted files (multiple file types and key types). (R-mode)	IV	17–43
ASCSRT	Synonym for ASCS$$. See above.		
AS$LIN	Return asynchronous line number.	IV	8–30
AS$LST	Retrieve asynchronous line characteristics.	IV	8–25
ASNLN$	Assign AMLC line.	IV	8–20
AS$SET	Set asynchronous line characteristics.	IV	8–31
ASSUR$	Check process has given amount of time slice left.	III	2–28
AT$	Set the attach point to a directory specified by pathname.	II	3–3

AT$ABS	Set the attach point to a specified top-level directory and partition.	II	3–6
AT$ANY	Set the attach point to a specified top-level directory on any partition.	II	3–9
AT$HOM	Set the attach point to the home directory.	II	3–11
AT$LDEV	Set the attach point by top-level directory and logical disk number.	II	3–13
AT$OR	Set the attach point to the login directory.	II	3–15
AT$REL	Set the attach point relative to the current directory.	II	3–17
AT$ROOT	Set the attach point to the root directory	II	3–19
ATCH$$	Set the attach point to a specified directory.	II	A–3
ATTDEV	Change a device assignment temporarily.	IV	3–5
BIN$SR	Perform binary search in ordered table.	III	6–21
BNSRCH	Binary search.	IV	17–49
BREAK$	Inhibit or enable BREAK function.	III	3–55
BUBBLE	Bubble sort.	IV	17–51
C$xy series	FORTRAN compiler conversion functions.	I	B–5
C$A01	Control functions for user terminal.	IV	6–4
C$M05	Control functions for 9-track tape.	IV	D–10
C$M10	Control functions for 7-track tape.	IV	D–10
C$M11	Control functions for 7-track tape (BCD).	IV	D–10
C$M13	Control functions for 9-track tape (EBCDIC).	IV	D–10
C$P02	Control functions for paper tape.	IV	6–11
C1IN	Read a character.	III	3–5
C1IN$	Read a character.	III	3–6
C1NE$	Read a character, suppressing echo.	III	3–7
CALAC$	Determine whether an object is accessible for a given action.	II	2–16
CASE$A	Convert between uppercase and lowercase.	IV	14–2

CAT$DL	Delete an access category.	II	2–18
CE$BRD	Return caller's maximum command environment breadth.	II	6–2
CE$DPT	Return caller's maximum command environment depth.	II	6–3
CF$EXT	Extend or truncate a CAM file.	II	4–132
CF$REM	Get a CAM file's extent map.	II	4–134
CF$SME	Set a CAM file's allocation size value.	II	4–137
CH$FX1	Convert string (decimal) to 16–bit integer.	III	6–3
CH$FX2	Convert string (decimal) to 32–bit	III	6–5
CH$HX2	Convert string (hexadecimal) to 32–bit integer.	III	6–7
CH$MOD	Change the open mode of an open file.	II	4–6
CH$OC2	Convert string (octal) to 32–bit integer.	III	6–9
CHG$PW	Change login validation password.	III	2–29
CKDYN$	Determine if routine is dynamically accessible.	III	2–4
CL$FNR	Close a file by name and return a bit string indicating closed units.	II	4–7
CL$GET	Read a line.	III	3–8
CL$MSG	Return text of specified system prompt.	III	2–5
CL$PIX	Parse command line according to a command line picture.	II	6–4
CLINEQ	Solve linear equations (complex).	IV	18–7
CLNU$S	Close all sort units after SRTF$.	IV	17–29
CLO$FN	Close a file system object by pathname.	II	4–9
CLO$FU	Close a file system object by file unit number.	II	4–10
CLOS$A	Close a file.	IV	15–2
CMADD	Matrix addition (complex).	IV	18–9
CMADJ	Calculate adjoint matrix (complex).	IV	18–11
CMBN$S	Sort tables prepared by SETU$.	IV	17–27
CMCOF	Calculate signed cofactor (complex).	IV	18–13
CMCON	Set constant matrix (complex).	IV	18–15
CMDET	Calculate matrix determinant (complex).	IV	18–17
CMDL$A	Parse a command line.	IV	16–2

CMIDN	Set matrix to identity matrix (complex).	IV	18–19
CMINV	Calculate signed cofactor (complex).	IV	18–21
CMLV$E	Call new command level after an error.	III	5–5
CMMLT	Matrix multiplication (complex).	IV	18–23
CMSCL	Multiply matrix by scalar (complex).	IV	18–25
CMSUB	Matrix subtraction (complex).	IV	18–27
CMTRN	Calculate transpose matrix (complex).	IV	18–29
CNAM$$	Change the name of an object in the current directory.	II	4–11
CNIN$	Read a specified number of characters.	III	3–11
CNSIG$	Continue scan for on-units.	III	7–20
CNVA$A	Convert ASCII number to binary.	IV	14–4
CNVB$A	Convert binary number to ASCII.	IV	14–6
CO$GET	Return information about command output settings.	III	3–56
COM$AB	Expand a line using Abbreviations preprocessor.	III	2–31
COMANL	Read a line into a PRIMOS buffer.	III	3–13
COMB	Generate matrix combinations.	IV	18–5
COMI$$	Switch input between the terminal and a file.	III	3–57
COMLV$	Call a new command level.	III	5–6
COMO$$	Switch output between the terminal and a file.	III	3–58
CONTRL	Perform device-independent control functions.	IV	4–11
CP$	Invoke a command from a running program.	II	6–8
CPUID$	Return model number of Prime computer.	III	2–7
CREA$$	Create a new subdirectory in the current directory.	II	A–5
CREPW$	Create a new password directory.	II	A–7
CSTR$A	Compare two strings for equality.	IV	10–3
CSUB$A	Compare two substrings for equality.	IV	10–5
CTIM$A	Return CPU time since login.	IV	12–2
CUS$CHANGE_ADMIN	Change user ID of the System Administrator.	IV	19–8
CUS$CHANGE_SYSTEM	Enable changes to the system attributes.	IV	19–10
CUS$CHECK_GROUP	Check if ACL group is on a system or a project.	IV	19–16

CUS$CHECK_PROJECT_ID	Check if a project is on the system.	IV	19–18
CUS$CHECK_SAD	Check SAD hashing status for system or project.	IV	19–19
CUS$CHECK_USER_ID	Check if user is on system or member of project.	IV	19–22
CUS$CLOSE_SAD	Close a System Administration Directory (SAD).	IV	19–24
CUS$CREATE_SAD	Create a System Administration Directory (SAD).	IV	19–25
CUS$GROUP	Add an ACL group to the SAD.	IV	19–29
CUS$LIST_GROUP_NAMES	List the system and project ACL groups.	IV	19–31
CUS$LIST_GROUPS_PROJECTS	List the projects using an ACL group.	IV	19–33
CUS$LIST_GROUPS_USERS	List users of a system or project ACL group.	IV	19–36
CUS$LIST_PROJECT	List the attributes of a specific project.	IV	19–39
CUS$LIST_PROJECT_NAMES	List the projects on the system.	IV	19–43
CUS$LIST_SYSTEM	List the attributes of the system.	IV	19–45
CUS$LIST_USER	List the attributes of a specified user.	IV	19–50
CUS$LIST_USER_NAMES	List the users on the system or project.	IV	19–55
CUS$OPEN_SAD	Open a System Administration Directory (SAD).	IV	19–58
CUS$PROJECT	Add, delete, or change a specific project.	IV	19–61
CUS$REBUILD_SAD	Rebuild a SAD for a system or project.	IV	19–67
CUS$SA_MODE	Check if user is System Administrator of the SAD.	IV	19–69
CUS$USER	Add, delete, or change a specific user.	IV	19–70
CUS$VERIFY_USER	Check network for valid user ID on other systems.	IV	19–76
CV$DQS	Convert binary date to quadseconds.	III	6–12
CV$DTB	Convert ASCII date to binary format.	III	6–13
CV$FDA	Convert binary date to ISO format.	III	6–15
CV$FDV	Convert binary date to visual format.	III	6–17
CV$QSD	Convert quadsecond date to binary format.	III	6–19
D$xy series	FORTRAN compiler division functions.	I	B–8
D$INIT	Initialize disk.	IV	D–2
DATE$	Return current date and time.	III	2–11
DATE$A	Return current date, American style.	IV	12–3

DELE$A	Delete a file.	IV	15–3
DIR$CR	Create a new directory.	II	4–15
DIR$LS	Search for specified types of entries in a directory open on a file unit.	II	4–17
DIR$RD	Read sequentially the entries of a directory open on a file unit.	II	4–24
DIR$SE	Return directory entries meeting caller–specified selection criteria.	II	4–29
DISPLY	Update sense light settings (obsolete).	III	D–2
DKGEO$	Register disk format with driver.	IV	5–3
DLINEQ	Solve a system of linear equations (double precision).	IV	18–7
DMADD	Matrix additions (double precision).	IV	18–9
DMADJ	Calculate adjoint matrix (double precision).	IV	18–11
DMCOF	Calculate signed cofactor (double precision).	IV	18–13
DMCON	Set matrix to constant matrix (double precision).	IV	18–15
DMDET	Calculate determinant (double precision).	IV	18–17
DMIDN	Set matrix to identity matrix (double precision).	IV	18–19
DMINV	Calculate inverted matrix (double precision).	IV	18–21
DMMLT	Matrix multiplication (double precision).	IV	18–23
DMSCL	Multiply matrix by a scalar (double precision).	IV	18–25
DMSUB	Matrix subtraction (double precision).	IV	18–27
DMTRN	Calculate transpose matrix (double precision).	IV	18–29
DOFY$A	Return today's date as day of year (Julian).	IV	12–4
DS$AVL	Return data about a disk partition.	III	2–61
DS$ENV	Return data about a process's environment.	III	2–63
DS$UNI	Return data about file units.	III	2–67
DS$SEND_CUSTOMER_UM	Send a message to the DMS server	III	2–12
DTIM$A	Return disk time since login.	IV	12–5
DUPLX$	Control the way PRIMOS treats the user terminal.	III	3–60
DY$SGS	Return maximum number of dynamic segments.	III	4–24

E$xy series	FORTRAN compiler exponentiation routines.	I	B–8
ECL$CC	Supervise editing of input from terminal or command file (callable from C).	III	3–14
ECL$CL	Interface to ECL$CC (for non-C programs).	III	3–17
EDAT$A	Today's date, European (military) style.	IV	12–6
ENCD$A	Convert a numeric value to FORTRAN (printable) format.	IV	14–8
ENCRYPT$	Encrypt login validation passwords.	III	6–23
ENT$RD	Return the contents of a named entry in a directory open on a file unit.	II	4–38
EPF$AL	Perform the linkage allocation phase for an EPF.	II	5–3
EPF$ALLC	Perform the linkage allocation phase for an EPF.	II	5–3
EPF$CP	Return the state of the command processing flags in an EPF.	II	5–5
EPF$CPF	Return the state of the command processing flags in an EPF.	II	5–5
EPF$DEL	Deactivate the most recent invocation of a specified EPF.	II	5–7
EPF$DL	Deactivate the most recent invocation of a specified EPF.	II	5–7
EPF$INIT	Perform the linkage initialization phase for an EPF.	II	5–9
EPF$NT	Perform the linkage initialization phase for an EPF.	II	5–9
EPF$INVK	Initiate the execution of a program EPF.	II	5–11
EPF$VK	Initiate the execution of a program EPF.	II	5–11
EPF$ISREADY	Indicate whether a registered EPF is ready or suspended.	II	5–15
EPF$MAP	Map the procedure images of an EPF file into virtual memory.	II	5–17
EPF$MP	Map the procedure images of an EPF file into virtual memory.	II	5–17
EPF$REG	Register an EPF.	II	5–20
EPF$RN	Combine functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.	II	5–22
EPF$RUN	Combine functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.	II	5–22
EPF$UREG	Unregister an EPF.	II	5–25
EQUAL$	Generate a filename based on another name.	II	4–40

ERKL$$	Read or set the erase and kill characters.	III	3–63
ER$PNT	Print error messages on terminal (FTN).	III	3–34
ER$PRINT	Print error messages on terminal.	III	3–34
ERRPR$	Print a standard error message (obsolete).	III	D–3
ERRSET	Set ERRVEC (a system error vector) (obsolete).	III	D–5
ER$TEXT	Return error message to a variable.	III	2–15
ER$TXT	Return error message to a variable (FTN).	III	2–15
ERTXT$	Return text associated with error code (obsolete).	III	D–7
EX$CLR	Disable signalling of EXIT$ condition.	III	7–36
EX$RD	Return state of EXIT$ signalling.	III	7–37
EX$SET	Enable signalling of EXIT$ condition.	III	7–38
EXIT	Return to PRIMOS.	III	5–7
EXST$A	Check for file existence.	IV	15–4
EXTR$A	Return an object's entryname and parent directory pathname.	II	4–41
F$xxyy series	FORTRAN compiler floating-point functions.	I	B–8
FDAT$A	Convert the DATMOD field returned by RDEN$$ to DAY MON DD YYYY.	IV	14–10
FEDT$A	Convert the DATMOD field returned by RDEN$$ to DAY DD MON YYYY.	IV	14–11
FIL$DL	Delete a file identified by a pathname.	II	4–43
FILL$A	Fill a string with a character.	IV	10–7
FINFO$	Return information about a specified file unit.	II	4–45
FNCHK$	Verify a supplied string as a valid filename.	II	4–49
FORCEW	Force PRIMOS to write modified records to disk.	II	4–51
FRE$RA	Deallocate space for EPF function return information.	III	4–22
FSUB$A	Fill a substring with a specified character.	IV	10–9
FTIM$A	Convert the TIMMOD field returned by REDN$$.	IV	14–12

G$METR	Return system metering information.	III	2–72
GCHAR	Get a character from an array.	III	6–24
GCHR$A	Get a character from a packed string.	IV	10–11
GEND$A	Position to end of file.	IV	15–5
GEN$PW	Generate a login validation password.	III	2–32
GETERR	Return ERRVEC contents (obsolete).	III	D–8
GETID$	Obtain the user ID and the groups to which it belongs.	II	2–19
GINFO	Return PRIMOS II information.	III	2–17
GPAS$$	Obtain the passwords of a subdirectory of the current directory.	II	2–21
GPATH$	Return the pathname of a specified unit, attach point, or segment.	II	4–53
GSNAM$	Return current PRIMOS system name.	III	2–19
GT$PAR	Parse character string into tokens.	III	6–25
GTROB$	Find out whether current attach point is on a robust partition.	II	3–21
GV$GET	Retrieve the value of a global variable.	II	6–11
GV$SET	Set the value of a global variable.	II	6–13
H$xy series	FORTRAN compiler complex number storage.	I	B–5
HEAP	Heap sort.	IV	17–52
I$AA01	Read ASCII from terminal.	IV	6–7
I$AA12	Read ASCII from terminal or input stream by REDN$$.	IV	6–9
I$AC03	Input from parallel card reader.	IV	7–26
I$AC09	Input from serial card reader.	IV	7–28
I$AC15	Read and print card from parallel card reader.	IV	7–30
I$AD07	Read ASCII from disk.	IV	5–4
I$AM05	Read ASCII from 9-track tape.	IV	D–12
I$AM10	Read ASCII from 7-track tape.	IV	D–12
I$AM11	Read BCD from 7-track tape.	IV	D–12

I$AM13	Read EBCDIC from 9-track tape.	IV	D–12
I$AP02	Read paper tape (ASCII).	IV	6–12
I$BD07	Read binary from disk.	IV	5–6
I$BM05	Read binary from 9-track.	IV	D–12
I$BM10	Read binary from 7-track.	IV	D–12
ICE$	Initialize the command environment.	III	5–8
IDCHK$	Validate a name.	III	2–33
IMADD	Matrix addition (integer).	IV	18–9
IMADJ	Calculate adjoint matrix (integer).	IV	18–11
IMCOF	Calculate signed cofactor (integer).	IV	18–13
IMCON	Set matrix to constant matrix (integer).	IV	18–15
IMDET	Calculate matrix determinant (integer).	IV	18–17
IMIDN	Set matrix to identity matrix (integer).	IV	18–19
IMMLT	Matrix multiplication (integer).	IV	18–23
IMSCL	Multiply matrix by scalar (integer).	IV	18–25
IMSUB	Matrix subtraction (integer).	IV	18–27
IMTRN	Calculate transpose matrix (integer).	IV	18–29
IN$LO	Determine if a forced logout is in progress.	III	2–34
INSERT	Insertion sort.	IV	17–53
IOA$	Provide free-format output.	III	3–36
IOA$ER	Provide free-format output, for error messages.	III	3–43
IOA$RS	Perform free-format output to a buffer.	III	6–30
IOCS$F	Free logical unit.	IV	3–4
IOCS$_FREE_LOGICAL_UNIT	Free logical unit.	IV	3–4
IOCS$G	Get logical unit.	IV	3–2
IOCS$_GET_LOGICAL_UNIT	Get logical unit.	IV	3–2
ISACL$	Determine whether an object is ACL-protected.	II	2–23
IS$AB	Allocate an ISC message buffer.	V	10–5
IS$AS	Accept an ISC session.	V	8–9
IS$CE	Clear an ISC session exception.	V	11–7

IS$FB	Free an ISC message buffer.	V	10–7
IS$GE	Get an ISC session exception.	V	11–5
IS$GRQ	Get an ISC session request.	V	8–6
IS$GRS	Get an ISC session request response.	V	8–12
IS$GSA	Get ISC session attributes.	V	14–4
IS$GSO	Get list of ISC sessions owned by this server.	V	14–2
IS$GSS	Get ISC session status information.	V	14–7
IS$RM	Receive an ISC message.	V	10–12
IS$RS	Request an ISC session.	V	8–3
IS$SM	Send an ISC message.	V	10–9
IS$STA	Get ISC current session statistics.	V	14–10
IS$TS	Terminate an ISC session.	V	11–3
ISN$C	Catalog ISC server's Low Level Name.	V	7–5
ISN$L	Look up ISC server's Low Level Name.	V	7–7
ISN$RC	Recatalog ISC server's Low Level Name File.	V	7–8
ISN$UC	Uncatalog (delete) ISC server's Low Level Name.	V	7–9
ISREM$	Determine whether an open file system object is local or remote.	II	4–56
JSTR$A	Left-justify, right-justify, or center a string.	IV	10–13
KLM$IF	Get serialization data about Prime software.	III	5–10
L$xy series	FORTRAN compiler complex number loading.	I	B–5
LDISK$	Return information on the system's disk table.	II	4–58
LIMIT$	Set and read various timers.	III	8–30
LINEQ	Solve a system of linear equations (single precision).	IV	18–7
LIST$CMD	Return a list of commands valid at mini-command level.	II	6–15
LN$SET	Modify user's search rules to permit dynamic linking to EPF library.	II	5–27

LOGO$$	Log out a user.	III	2–35
LON$CN	Switch logout notification on or off.	III	5–24
LON$R	Read logout notification information.	III	5–25
LOV$SW	Indicate if the login-over-login function is currently permitted.	III	2–20
LSTR$A	Locate one string within another.	IV	10–15
LSUB$A	Locate one substring within another.	IV	10–17
LUDEV$	Return a list of devices that a user can access.	III	2–37
LUDSK$	List the disks a given user is using.	II	4–61
LV$GET	Retrieve the value of a CPL local variable.	II	6–17
LV$SET	Set the value of a CPL local variable.	II	6–19
M$xy series	FORTRAN compiler multiplication routines.	I	B–8
MADD	Matrix addition (single precision).	IV	18–9
MADJ	Calculate adjoint matrix (single precision).	IV	18–11
MCHR$A	Move a character from one packed string to another.	IV	10–19
MCOF	Calculate signed cofactor (single precision).	IV	18–13
MCON	Set matrix to constant matrix (single precision).	IV	18–15
MDET	Calculate matrix determinant (single precision).	IV	18–17
MGSET$	Set the receiving state for messages.	III	9–4
MIDN	Set matrix to identity matrix (single precision).	IV	18–19
MINV	Calculate inverted matrix (single precision).	IV	18–21
MKLB$F	Convert FORTRAN statement label to PL/I format.	III	7–21
MKON$F	Create an on–unit (for FTN users).	III	7–22
MKON$P	Create an on–unit (for any language except FTN).	III	7–24
MKONU$	Create an on–unit (for PMA and PL/I users).	III	7–26
MM$MLPA	Make the last page of a segment available.	III	4–5
MM$MLPU	Make the last page of a segment unavailable.	III	4–6
MMLT	Matrix multiplication (single precision).	IV	18–23
MOVEW$	Move a block of memory.	III	6–32

MRG1$S	Merge sorted files.	IV	17–34
MRG2$S	Return next merged record.	IV	17–38
MRG3$S	Close merged input files.	IV	17–39
MSCL	Matrix addition (single precision).	IV	18–25
MSG$ST	Return the receiving state of a user.	III	9–2
MSTR$A	Move one string to another.	IV	10–21
MSUB	Matrix subtraction (single precision).	IV	18–27
MSUB$A	Move one substring to another.	IV	10–23
MTRN	Calculate transpose matrix (single precision).	IV	18–29
N$xy series	FORTRAN compiler negation functions.	I	B–5
NAM$AD_PORTAL	Convert an existing directory into a portal.	II	4–63
NAM$L_GMT	List accessible partitions and portals.	II	4–65
NAM$RM_PORTAL	Delete a portal entry in the specified directory	II	4–68
NAMEQ$	Compare two character strings.	III	6–33
NLEN$A	Determine the operational length of a string.	IV	10–25
NT$LTS	Return characteristics of PRIMOS network terminal service line.	IV	8–34
O$AA01	Write ASCII to terminal or command stream.	IV	6–5
O$AC03	Parallel interface to card punch.	IV	7–31
O$AC15	Parallel interface card punch and print.	IV	7–32
O$AD07	Write compressed ASCII to disk.	IV	D–3
O$AD08	Write ASCII uncompressed to disk.	IV	5–7
O$ALxx	Interface to various printer controllers.	IV	7–4
O$AL04	Centronics line printer.	IV	7–3
O$AL06	Parallel interface to MPC line printer.	IV	7–3
O$AL14	Versatec printer/plotter interface.	IV	7–18
O$AM05	Write ASCII to 9-track tape.	IV	D–12
O$AM10	Write ASCII to 7-track tape.	IV	D–12

O$AM11	Write BCD to 7-track tape.	IV	D–12
O$AM13	Write EBCDIC to 9-track tape.	IV	D–12
O$BD07	Write binary to disk.	IV	5–9
O$BM05	Write binary to 9-track tape.	IV	D–12
O$BM10	Write binary to 7-track tape.	IV	D–12
O$BP02	Punch paper tape (binary).	IV	6–14
OPEN$A	Open file specified by filename.	IV	15–6
OPNP$A	Read filename and open.	IV	15–8
OPNV$A	Open filename with verification and delay.	IV	15–10
OPSR$	Locate a file using a search list and open the file.	II	7–3
OPSRS$	Locate a file using a search list and a list of suffixes.	II	7–9
OPVP$A	Read filename and open, or verify and delay.	IV	15–13
OVERFL	Check if an overflow condition has occurred (obsolete).	III	D–9
P1IB	Input character from paper tape reader to Register A.	IV	6–16
P1IN	Input character from paper tape to variable.	IV	6–18
P1OB	Output character from Register A to paper-tape punch.	IV	6–17
P1OU	Output character from variable to paper-tape punch.	IV	6–19
PA$DEL	Remove an object's priority access.	II	2–24
PA$LST	Obtain the contents of an object's priority ACL.	II	2–25
PA$SET	Set priority access on an object.	II	2–27
PAR$RV	Return a logical value indicating ACL and quota support.	II	4–69
PERM	Generate matrix permutations.	IV	18–31
PHANT$	Start a phantom process (obsolete).	III	D–10
PHNTM$	Start a phantom process.	III	5–27
PL1$NL	Perform a nonlocal GOTO.	III	7–28
POSN$A	Position in a file.	IV	15–16
PRERR	Print an error message (obsolete).	III	D–11
PRI$RV	Return operating system revision number.	III	2–22
PRJID$	Return the user's project identifier.	III	2–40

PRWF$$	Read, write, position, or truncate a file.	II	4–71
PTIME$	Return amount of CPU time used since login.	III	2–41
PWCHK$	Validate syntax of a password.	III	2–42
Q$READ	Return directory quota and disk record usage information.	II	4–79
Q$SET	Set a quota on a subdirectory of the current directory.	II	4–82
QUICK	Partition exchange sort.	IV	17–54
QUIT$	Determine if there are pending quits.	III	3–65
RADXEX	Radix exchange sort.	IV	17–55
RAND$A	Generate random number and update seed, using 32-bit word size and the linear congruential method.	IV	13–2
RD$CE_DP	Return caller's current command environment breadth.	II	6–21
RD$CED	Return caller's current command environment breadth.	II	6–21
RDASC	Read ASCII from any device.	IV	4–5
RDBIN	Read binary from any device.	IV	4–9
RDEN$$	Position in or read from a directory.	II	A–8
RDLIN$	Read a line of characters from a compressed ASCII disk file.	II	4–85
RDTK$$	Parse a command line.	III	3–22
READY$	Display PRIMOS command prompt.	III	2–43
RECYCL	Tell PRIMOS to cycle to the next user (obsolete).	III	D–14
REMEPF$	Remove an EPF from a user's address space.	II	5–29
REST$$	Restore an R–mode executable image.	III	5–18
RESU$$	Restore and resume an R-mode executable image.	III	5–20
RLSE$S	Get input records after SETU$.	IV	17–26
RMSGD$	Receive a deferred message.	III	9–6
RNAM$A	Prompt, read a pathname, and check format.	IV	11–2
RNDI$A	Initialize random number generator seed.	IV	13–4
RNUM$A	Prompt and read a number (in any format).	IV	11–4

RPL$	Replace one EPF runfile with another.	II	5–31
RPOS$A	Return position of file.	IV	15–17
RRECL	Read disk record.	IV	D–5
RSEGAC$	Determine access to a segment.	III	2–23
RSTR$A	Rotate string left or right.	IV	10–26
RSUB$A	Rotate substring left or right.	IV	10–29
RTRN$S	Get sorted records.	IV	17–28
RVON$F	Revert an on-unit (for FTN users).	III	7–29
RVONU$	Revert an on-unit (for any language except FTN).	III	7–30
RWND$A	Reposition file.	IV	15–18
S$xy series	FORTRAN compiler subtraction routines.	I	B–8
SATR$$	Set or modify an object's attributes.	II	4–87
SAVE$$	Save an R-mode executable image.	III	5–21
SCHAR	Store a character into an array location.	III	6–35
SEM$CL	Release (close) a named semaphore.	III	8–16
SEM$DR	Drain a semaphore.	III	8–17
SEM$NF	Notify a semaphore.	III	8–18
SEM$OP	Open a set of named semaphores.	III	8–20
SEM$OU	Open a set of named semaphores.	III	8–20
SEM$TN	Periodically notify a semaphore.	III	8–24
SEM$TS	Return number of processes waiting on a semaphore.	III	8–26
SEM$TW	Wait on a specified named semaphore, with timeout.	III	8–27
SEM$WT	Wait on a specified named semaphore.	III	8–28
SETRC$	Record command error status.	III	5–14
SETU$S	Prepare sort table and buffers for CMBN$.	IV	17–22
SGD$DL	Delete a segment directory.	II	4–92
SGD$EX	Find out if there is a valid entry at the current position within the segment directory on a specified unit.	II	4–93
SGD$OP	Open a segment directory entry.	II	4–94

SGDR$$	Position, read, or modify a segment directory.	II	4–96
SGNL$F	Signal a condition.	III	7–31
SHELL	Diminishing increment sort.	IV	17–56
SID$GT	Return user number of initiating process.	III	2–44
SIGNL$	Signal a condition.	III	7–33
SIZE$	Return the size of a file system entry.	II	4–102
SLEEP$	Suspend a process for a specified interval.	III	8–34
SLEP$I	Suspend a process (interruptible).	III	8–35
SLITE	Set the sense light on or off (obsolete).	III	D–15
SLITET	Test sense light settings (obsolete).	III	D–16
SMSG$	Send an interuser message.	III	9–8
SNCHK$	Check validity of system name passed to it.	III	2–25
SP$REQ	Insert a file into the spool queue.	IV	7–12
SPAS$$	Set the owner and nonowner passwords on an object.	II	2–29
SPOOL$	Insert a file into the spool queue.	IV	7–10
SR$ABS	Disable optional rules enabled by SR$ENABL.	II	7–16
SR$ABSDS	Disable optional rules enabled by SR$ENABL.	II	7–16
SR$ADB	Add a rule to the start of a search list or before a specified rule within the list.	II	7–19
SR$ADDB	Add a rule to the start of a search list or before a specified rule within the list.	II	7–19
SR$ADDE	Add a rule to the end of a search list or after a specified rule within the list.	II	7–22
SR$ADE	Add a rule to the end of a search list or after a specified rule within the list.	II	7–22
SR$CRE	Create a search list.	II	7–25
SR$CREAT	Create a search list.	II	7–25
SR$DEL	Delete a search list.	II	7–27
SR$DSA	Disable an optional search rule enabled by SR$ENABL.	II	7–29
SR$DSABL	Disable an optional search rule enabled by SR$ENABL.	II	7–29
SR$ENA	Enable an optional search rule.	II	7–32

SR$ENABL	Enable an optional search rule.	II	7–32
SR$EXS	Determine if a search rule exists.	II	7–35
SR$EXSTR	Determine if a search rule exists.	II	7–35
SR$FR_LS	Free list structure space allocated by SR$LIST or SR$READ.	II	7–39
SR$FRL	Free list structure space allocated by SR$LIST or SR$READ.	II	7–39
SR$INI	Initialize all search lists to system defaults.	II	7–41
SR$INIT	Initialize all search lists to system defaults.	II	7–41
SR$LIS	Return the names of all defined search lists.	II	7–43
SR$LIST	Return the names of all defined search lists.	II	7–43
SR$NEX	Read the next rule from a search list.	II	7–47
SR$NEXTR	Read the next rule from a search list.	II	7–47
SR$REA	Read all of the rules in a search list.	II	7–52
SR$READ	Read all of the rules in a search list.	II	7–52
SR$REM	Remove a rule from a search list.	II	7–56
SR$SET	Set the locator pointer for a search rule.	II	7–59
SR$SETL	Set the locator pointer for a search rule.	II	7–59
SR$SSR	Set a search list via a user-defined search rules file.	II	7–62
SRCH$$	Open, close, delete, or verify existence of an object.	II	4–105
SRSFX$	Search for a file with a list of possible suffixes.	II	4–114
SRS$GN	Get server name.	V	7–10
SRS$GP	Get process numbers of all processes that have the same server name.	V	7–11
SRS$LN	List all active ISC server names.	V	7–13
SRTF$S	Sort several input files.	IV	17–16
SS$ERR	Signal an error in a subsystem.	III	5–16
SSTR$A	Shift string left or right.	IV	10–31
SSUB$A	Shift substring left or right.	IV	10–33
SSWTCH	Test sense switch settings (obsolete).	III	D–17
ST$SGS	Return maximum number of static segments.	III	4–25

STR$AL	Allocate user-class dynamic memory.	III	4–7
STR$AP	Allocate process-class dynamic memory.	III	4–8
STR$AS	Allocate subsystem-class dynamic memory.	III	4–9
STR$AU	Allocate user-class dynamic memory.	III	4–10
STR$FP	Free process-class dynamic memory.	III	4–11
STR$FR	Free user-class dynamic memory.	III	4–12
STR$FS	Free subsystem-class dynamic memory.	III	4–13
STR$FU	Free user-class dynamic memory.	III	4–14
SUBSRT	Sort file on ASCII key. (V-mode)	IV	17–10
SUBSRT	Sort file on ASCII key. (R-mode)	IV	17–41
SUSR$	Test if current user is supervisor.	III	2–45
SYN$CHCK	Return total of notices or waiters on a synchronizer.	V	4–2
SYN$CK	Return total of notices or waiters on a synchronizer (FTN).	V	4–2
SYN$CR	Create an event synchronizer (FTN).	V	2–5
SYN$CREA	Create an event synchronizer.	V	2–5
SYN$DE	Destroy an event synchronizer (FTN).	V	2–15
SYN$DEST	Destroy an event synchronizer.	V	2–15
SYN$GC	Create an event group (FTN).	V	3–5
SYN$GCHK	Return total of notices or waiters on an event group.	V	4–4
SYN$GCRE	Create an event group.	V	3–5
SYN$GD	Destroy an event group (FTN).	V	3–18
SYN$GDST	Destroy an event group.	V	3–18
SYN$GK	Return total of notices or waiters on an event group (FTN).	V	4–4
SYN$GL	List total of groups in server and their identifiers (FTN).	V	4–12
SYN$GLST	List total of groups in server and their identifiers.	V	4–12
SYN$GR	Retrieve a notice from a group (FTN).	V	3–15
SYN$GRTR	Retrieve a notice from a group.	V	3–15
SYN$GT	Perform a timed wait on a group (FTN).	V	3–13

SYN$GTWT	Perform a timed wait on a group.	V	3–13
SYN$GW	Wait on an event group (FTN).	V	3–11
SYN$GWT	Wait on an event group.	V	3–11
SYN$IF	Return information about a synchronizer (FTN).	V	4–6
SYN$INFO	Return information about a synchronizer.	V	4–6
SYN$LG	List total of synchronizers in group and their identifiers (FTN).	V	4–8
SYN$LIST	List total of synchronizers in server and their identifiers.	V	4–10
SYN$LS	List total of synchronizers in server and their identifiers (FTN).	V	4–10
SYN$LSIG	List total of synchronizers in group and their identifiers.	V	4–8
SYN$MV	Move a synchronizer into a group (FTN).	V	3–7
SYN$MVTO	Move a synchronizer into a group.	V	3–7
SYN$PO	Post a notice on a synchronizer (FTN).	V	2–7
SYN$POST	Post a notice on a synchronizer.	V	2–7
SYN$REMV	Remove a synchronizer from a group.	V	3–9
SYN$RM	Remove a synchronizer from a group (FTN).	V	3–9
SYN$RTRV	Retrieve a notice from an event synchronizer.	V	2–13
SYN$RV	Retrieve a notice from an event synchronizer (FTN).	V	2–13
SYN$TMWT	Perform a timed wait on an event synchronizer.	V	2–11
SYN$TW	Perform a timed wait on an event synchronizer (FTN).	V	2–11
SYN$WAIT	Wait on an event synchronizer.	V	2–9
SYN$WT	Wait on an event synchronizer (FTN).	V	2–9
T$AMLC	Communicate with AMLC driver.	IV	8–22
T$CMPC	Input from MPC card reader.	IV	7–33
T$LMPC	Move data to MPC line printer.	IV	7–15
T$MT	Raw data mover for tape.	IV	7–38
T$PMPC	Raw data mover for card reader.	IV	7–35
T$SLC0	Communicate with SMLC driver.	IV	8–3

T$VG	Interface to Versatec printer.	IV	7–21
T1IB	Read a character (function) from PMA into Register A.	III	3–28
T1IN	Read a character (procedure).	III	3–29
T1OB	Write one character from Register A.	III	3–52
T1OU	Write one character.	III	3–53
TEMP$A	Open a scratch file.	IV	15–19
TEXTO$	Check filename for valid format (obsolete).	III	D–18
TI$MSG	Display standard message showing times used.	III	2–46
TIDEC	Read a decimal number.	III	3–30
TIHEX	Read a hexadecimal number.	III	3–31
TIMDAT	Return timing information and user identification.	III	2–47
TIME$A	Return time of day.	IV	12–7
TIOCT	Read an octal number.	III	3–32
TL$SGS	Return highest segment number.	III	4–26
TMR$CANL	Cancel a timer.	V	5–15
TMR$CN	Cancel a timer (FTN).	V	5–15
TMR$CR	Create a timer (FTN).	V	5–6
TMR$CREA	Create a timer.	V	5–6
TMR$DE	Destroy a timer (FTN).	V	5–8
TMR$DEST	Destroy a timer.	V	5–8
TMR$GINF	Return permanent time information.	III	2–49
TMR$GTIM	Return current system time.	III	2–51
TMR$GTMR	Return information about a timer.	V	5–16
TMR$IF	Return permanent time information (FTN).	III	2–49
TMR$LIST	List total number of timers in server and their identifiers.	V	5–19
TMR$LOCALCONVERT	Convert local time to Universal Time.	III	2–52
TMR$LS	List total number of timers in server and their identifiers (FTN).	V	5–19
TMR$LU	Convert local time to Universal Time (FTN).	III	2–52
TMR$SA	Set an absolute timer (FTN).	V	5–9

TMR$SABS	Set an absolute timer.	V	5–9
TMR$SI	Set an interval timer (FTN).	V	5–11
TMR$SINT	Set an interval timer.	V	5–11
TMR$SR	Set a repetitive timer (FTN).	V	5–13
TMR$SREP	Set a repetitive timer.	V	5–13
TMR$TI	Return information about a timer (FTN).	V	5–16
TMR$TM	Return current system time (FTN).	III	2–51
TMR$UL	Convert Universal Time to local time (FTN).	III	2–54
TMR$UNIVCONVERT	Convert Universal Time to local time.	III	2–54
TNCHK$	Verify a supplied string as a valid pathname.	II	4–121
TNOU	Write characters to terminal, followed by NEWLINE.	III	3–45
TNOUA	Write characters to terminal.	III	3–46
TODEC	Write a signed decimal number.	III	3–47
TOHEX	Write a hexadecimal number.	III	3–48
TONL	Write a NEWLINE.	III	3–49
TOOCT	Write an octal number.	III	3–50
TOVFD$	Write a decimal number, without spaces.	III	3–51
TREE$A	Test for a pathname.	IV	10–35
TRNC$A	Truncate a file.	IV	15–21
TSCN$A	Scan the file system tree structure.	IV	15–22
TSRC$$	Open, close, delete, or find a file anywhere in the file structure.	II	A–17
TTY$IN	Check for unread terminal input characters.	III	3–66
TTY$OUT	Check for characters in user's terminal input buffer.	III	3–67
TTY$RS	Clear the terminal input and output buffers.	III	3–68
TYPE$A	Determine string type.	IV	10–38
UID$BT	Return unique bit string.	III	6–37
UID$CH	Convert UID$BT output into character string.	III	6–38
UNIT$A	Check for file open.	IV	15–27

UNITS$	Return caller's minimum and maximum file unit numbers.	II	4–124
UNO$GT	List users with same name as caller.	III	2–56
UPDATE	Update current directory (PRIMOS II only) (obsolete).	III	D–20
USER$	Return user number and count of users.	III	2–26
UTYPE$	Return user type of current process.	III	2–57
VALID$	Validate a name against composite identification.	III	2–59
WILD$	Return a logical value indicating whether a wildcard name was matched.	II	4–125
WRASC	Write ASCII.	IV	4–3
WRBIN	Write binary to any output device.	IV	4–7
WRECL	Write disk record.	IV	D–8
WTLIN$	Write a line of characters to a compressed ASCII file.	II	4–126
YSNO$A	Ask question and obtain a yes or no answer.	IV	11–7
Z$80	Clear double-precision exponent.	I	B–5

Index

A

Access, changing, 4–105
Access category, deleting, 2–18
Access control
  See also Access
  ACL structure, 2–12
  converting ACL to password, 2–13
  copying another object's ACLs, 2–9
  definition, 2–1
  determining protection, 2–23
  group ID, 2–12
  partition support of, 4–69
  setting, 2–14
  user ID, 2–12
Access
  See also Access control
  calculating accessibility, 2–16
  category, 2–3, 2–15
  changing, 2–5
  default, 2–7
  listing, 2–11
  modifying, 2–5
  nonowner, 2–17
  nonowner passwords, 2–29
  owner, 2–17
  owner passwords, 2–29
  specific, 2–15
Access, priority
  obtaining object's contents, 2–25
  removing, 2–24
  setting, 2–27
ACLs. *See* Access; Access control
Addressing modes, subroutine libraries, 1–13
Arguments
  bits, position values, 1–11
  keys, 1–12
  parsing command, 6–4
  setting bits in, 1–10
Arrays, in declarations, 1–8
ASCII disk file, reading characters from, 4–85
Attach points
  any directory, 3–3
  current directory, 3–1
  definition, 3–1
  home directory, 3–11
  login directory, 3–15
  returning pathname of, 4–53
  root directory, 3–19
  set to top–level directory by ldev, 3–13
  subdirectory, 3–17
  top–level directory, 3–6, 3–9
Attaching, to file directory, 3–1
Attributes
  date/time last accessed, 4–90
  date/time modified, 4–90
  dumped bit, 4–90
  modifying, 4–87
  read/write lock, 4–90
  setting directory, 4–15
  setting object, 4–87

B

BIN data types, 1–7
BIT data types, 1–7
Bits
  position, values of, 1–11
  setting in arguments, 1–10

C

CALL statement
  declaration, 1–4
  function, 1–5
  functions called as subroutines, 1–10
CAM files
  allocation size values, 4–130, 4–137
  creating, 4–129
  defined, 4–128
  extending, 4–129
  extent maps, 4–129
  maximum extent size, 4–130
  maximum size of, 4–128
  minimum extent size, 4–130
  moving physical end of, 4–132
  physical layout on disk, 4–134
  structure, 4–128
  truncating, 4–129
Change open mode, 4–6
CHAR data types, 1–7
CL$PIX option arguments
  multiple instances of, C–9
  using repeat counts with, C–10
CL$PIX subroutine
  argument group, C–2
  arguments, default values, C–5
  calls, C–13
  command line options, –LISTING, C–7
  data types, C–2
  in CPL mode, C–10
  local variable area, C–11
  normal mode, C–1
  object argument groups, C–2
  operating modes, C–1
  option arguments, C–6
  punctuation in, C–2
  repeat counts, C–6
  REST data type options, C–7
  syntax differences, C–10
Closing file system objects, 4–7
  by pathname, 4–9
  by unit number, 4–10
Command arguments, parsing, 6–4

Command environment
 See also Commands
 current breadth, 6–21
 discussion, 6–1
 maximum breadth, 6–2
 setting maximum depth, 6–3
Command line parsing, using CL$PIX, example of, C–11
Command processor, discussion, 6–1
Command state flags, EPFs, 5–5
Commands
 See also Command environment
 invoking from programs, 6–8
 listing at mini–command level, 6–15
Common file system name space, definition, 4–22, 4–28, 4–66
Converting, directory entry to portal, 4–63
COPY command, –RWLOCK option, 4–91
CPL local variables
 retrieving values, 6–17
 setting, 6–19
CPL mode, CL$PIX subroutine, C–10
 usage differences, C–11
Creating
 CAM files, 4–129
 directories, 4–15

D

Data structures
 dir_entries, 4–33
 dir_entry, 4–22, 4–25, 4–28, 4–36
Data types
 BIN, 1–7
 BIT, 1–7
 CHAR, 1–7
 CL$PIX structure, C–4
 FLOAT BIN, 1–7
 for Prime languages, B–1
 parameters, 1–6
 POINTER, 1–7
 REST, C–5
 returned value, 1–6
 UNCL, C–5
Date/time last accessed attribute, 4–90
Date/time modified attribute, 4–90
Deactivating an EPF, 5–7
Declarations
 array, 1–8
 data types in a picture, C–4
 DECLARE statement, 1–4
 function, 1–4, 1–5
 structures, 1–8
 subroutines, 1–4
DECLARE statement, RETURN descriptor, 1–5
Default values, arguments in a picture, C–5
Delete access, 4–43
Deleting
 access category, 2–18
 files, 4–43
 portals, 4–68
 segment directories, 4–92
Dependency list, of registered EPFs, 5–21
dir_entry, valid attributes for root directory, 4–22, 4–28, 4–36
Directories
 See also Segment directories
 adding objects to, 4–109
 converting to portal, 4–63
 creating, 4–15
 deleting empty, 4–43
 listing entries, 4–17
 obtaining quota information, 4–79
 reading entries in, 4–24
 searching through, 4–17, 4–29
 setting attributes, 4–15
 setting quotas, 4–82
Directory passwords, obtaining, 2–21
Disk partitions, listing currently mounted, 4–65
Disk Record Availability (DSKRAT) file, indication of, 4–26
Disk Record Availability Table (DSKRAT), pool of free records, 4–43
Disk records, obtaining information on, 4–79
Disk table
 local disks, 4–61
 obtaining information on, 4–58
 use at Rev. 23.0, 4–61
Disks, listing by user, 4–61
Dumped bit, 4–90
Dynamic linking, to an library EPF, 5–27

E

EPFs
 See also Registered EPFs
 command state flags, 5–5
 deactivating, 5–7
 dynamic linking to library EPF, 5–27
 enabling registration, 5–20
 initializing, 5–9
 invoking program, 5–11
 linkage allocation, 5–3
 managing, 5–1
 mapping procedure images, 5–17
 ready or suspended status, 5–15
 removing from user's address space, 5–29
 replacing one with another, 5–31
Error codes
 in subroutine arguments, 1–12
 SYSCOM directory, 1–13
Extending CAM files, 4–129
Extent maps, definition, 4–129
Extents, 4–128

F

File system
 common file system name space, 4–22, 4–28, 4–66
 obsolete subroutines, A–1
File system objects
 changing access to, 4–105
 changing name of, 4–11
 closing, 4–7, 4–105
 closing by pathname, 4–9
 closing by unit number, 4–10
 creating CAM files, 4–109
 deleting, 4–105
 determining local or remote, 4–56
 opening, 4–105, 4–107
 returning size of, 4–102
 verifying, 4–105, 4–109
File units
 obtaining information about, 4–45
 returning numbers in use, 4–124

File units (Continued)
  returning pathname of, 4–53
Filenames
  extracting from pathname, 4–41
  generating, 4–40
  verifying, 4–49
Files
  See also CAM files; File system objects
  CAM, 4–128
  creating CAM, 4–109
  deleting, 4–43
  Disk Record Availability (DSKRAT), 4–26
  forced writing to disk, 4–51
  locating and opening subroutines, 7–9
  manipulating, 4–71
  using search list to locate and open, 7–5
  writing in compressed ASCII format, 4–126
FIXED BIN data types, 1–7
FLOAT BIN data types, 1–7
Forced writing to disk, 4–51
Functions
  CALL statement, 1–5
  declaration, 1–4
  defined, 1–1
  OPTIONAL RETURNED VALUE, 1–10
  RETURNED VALUE, 1–10
  subroutines, called as, 1–10

G

Global Mount Table
  propagating references, 4–54
  reading contents of, 4–65
  references to remote portals, 4–54
  remote disks, 4–61
  remote system names, 4–60
Global variables
  retrieving values, 6–11
  setting value, 6–13

I

Initializing, EPFs, 5–9
Invoking commands from programs, 6–8
Invoking program EPFs, 5–11

K

Keys, in arguments, 1–12

L

Linkage allocation, EPFs, 5–3
LIST_REGISTERED_EPF command, 5–25
Listing, directories, 4–17
Listing commands, at mini–command level, 6–15
Listing disks, by user, 4–61
-LISTING option, CL$PIX subroutine, command line options, C–7
Local file system object, determining, 4–56
Local variables. *See* CPL local variables
Locating files, search rule subroutines, 7–5
Login directory, setting attach point to, 3–15

M

MAGRST utility
  See also MAGSAV utility
  restoring read/write locks, 4–91
MAGSAV utility
  See also MAGRST utility
  saved by, 4–26
  saving read/write locks, 4–91
Messages, MAXIMUM QUOTA EXCEEDED, 4–83
Mini–command level, listing commands, 6–15
Modifying, object attributes, 4–87

N

Nonowner passwords, 2–29

O

Object names
  See also File system objects
  changing name of, 4–11
Omitting parameters, example of, 1–9
Open mode, changing, 4–6
Option arguments, CL$PIX, C–6
OPTIONAL RETURNED VALUES, functions, 1–10
Owner passwords, 2–29

P

Parameters
  data types, 1–6
  data types in, 1–6
  omitting, 1–9
  subroutines, 1–5
Parsing, command arguments, 6–4
Partitions
  ACL support of, 4–69
  listing currently mounted, 4–65
Passwords
  naming conventions, 2–29
  nonowner, 2–29
  obtaining directory, 2–21
  owner, 2–29
Pathnames
  appending suffixes, 4–4
  returning, 4–53
  verifying string as, 4–121
Picture syntax, CL$PIX, C–1
POINTER data types, 1–7
Portal, definition, 4–68
Portals
  affect on root directory, 4–22, 4–26, 4–28, 4–36
  converting directory to, 4–63
  definition, 4–64
  deleting, 4–68
  listing currently mounted, 4–65
  referencing remote systems, 4–64
Positioning, files, 4–71
Priority access. *See* Priority ACLs
Priority ACLs
  discussion, 2–27
  obtaining object's contents, 2–25
  removing, 2–24
  setting, 2–27
Programs, invoking commands from, 6–8

Q

Quotas
- obtaining information on, 4–79
- partition, 4–69
- setting on subdirectories, 4–82

R

Read/write locks
- meaning of bits, 4–90
- saving and restoring, 4–91
- with segment directories, 4–90

Reading, characters from ASCII disk file, 4–85
Reading, files, 4–71
Records
- *See also* Disk records
- forced writing to disk, 4–51

Registered EPFs
- *See also* EPFs
- database for, 5–21
- definition, 5–1
- dependency list of, 5–21
- enabling registration, 5–20
- status, 5–15
- unregistering, 5–25

Remote file system object, determining, 4–56
Removing, priority ACLs, 2–24
Removing EPFs, from user's address space, 5–29
Repeat counts, CL$PIX argument groups, C–6
Replacing EPFs, one EPF runfile with another, 5–31
REST argument, –STRING option with, C–8
REST data type, CL$PIX subroutine, C–5
RETURNED VALUE, functions, 1–10
Returned values, data types, 1–6
Returning, size of file system entry, 4–102
Returning pathnames, 4–53
RETURNS descriptor, DECLARE statement, 1–5
Robust partitions
- creating CAM files on, 4–109
- files, 4–128
- searching, 3–21

Root directory, 4–28, 4–36
- affect of EXTR$A, 4–42
- setting attach point to, 3–19
- valid attributes, 4–28, 4–36
- valid dir_entry attributes, 4–22, 4–28, 4–36

–RWLOCK option, COPY command, 4–91

S

Search rule subroutines
- adding to search list, 7–19, 7–22
- and arguments, 7–2
- and structures, 7–53
- creating blank search list, 7–25
- deleting search lists, 7–27
- determining search rule existence in search list, 7–35
- disabling optional search rules, 7–29
- disabling rules, 7–16
- enabling optional search rules, 7–32
- freeing allocated space, 7–39
- initializing search lists to system defaults, 7–41
- listed, 7–1
- locating and opening files, 7–3
- modifying the locator pointer, 7–59
- reading from search lists, 7–47, 7–52
- removing rules from search list, 7–56
- returning search list names, 7–43
- setting search lists, 7–62
- suffixes used to open files, 7–9

Search rules
- modifying for dynamic linking, 5–27
- optional, 7–17

Searching, directories, 4–17
Segment directories
- closing files within, 4–110
- creating, 4–110
- creating files within, 4–110
- definition, 4–110
- deleting, 4–92, 4–110
- deleting files within, 4–110
- entry validation, 4–93
- modifying size of, 4–96
- opening entries, 4–94
- opening files within, 4–110
- operations involving, 4–110
- positioning in, 4–96
- read/write locks, 4–90
- reading entry in, 4–96
- returning pathname of, 4–53

Setting
- CPL local variables, 6–19
- global variable values, 6–13
- maximum breadth of command environment, 6–2
- maximum depth of command environment, 6–3
- object attributes, 4–87
- priority ACLs, 2–24
- quotas on subdirectories, 4–82

Structures, in declarations, 1–8
Subdirectories, operations on, 4–109
Subroutines
- arguments, keys in, 1–12
- CALL statement, 1–4
- contents of, 1–2
- declaration, 1–4
- defined, 1–1
- error codes, 1–12
- example of, 1–3
- obsolete, A–1
- parameters, 1–5
- usage section, 1–2

Suffixes
- appending to pathname, 4–4
- searching for file, 4–114

SYSCOM directory, error codes, 1–13

T

Truncating
- CAM files, 4–129
- files, 4–71

U

UNCL data type
- CL$PIX subroutine, C–5
- with option arguments, C–8

Unregistering EPFs, 5–25
Usage section
- date types in, 1–6

Usage section (Continued)
  explained, 1–2
  parameters, 1–6
User ID, retrieving, 2–19

V

Validating, segment directory entries, 4–93
Values
  retrieving CPL local variable, 6–17
  retrieving global variable, 6–11
  setting CPL local variable, 6–19
Variables. *See* CPL local variables; Global variables
Verifying, filenames, 4–49

W

Wildcards, matching name with, 4–125
Writing files, 4–71
  in compressed ASCII format, 4–126
  to disk, forced, 4–51

Surveys

Reader Response Form
Subroutines Reference II: File System
DOC10081-2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

   □ *excellent* □ *very good* □ *good* □ *fair* □ *poor*

2. What features of this manual did you find most useful?

3. What faults or errors in this manual gave you problems?

4. How does this manual compare to equivalent manuals produced by other computer companies?

   □ *Much better* □ *Slightly better* □ *About the same*
   □ *Much worse* □ *Slightly worse* □ *Can't judge*

5. Which other companies' manuals have you read?

Name:__________
Position:__________
Company:__________
Address:__________

__________Postal Code:__________