



***Subroutines  
Reference Guide  
Volume II***

**DOC10081-1LA**

# **Subroutines Reference Guide Volume II**

**First Edition**

**by  
Len Bruns**

**Updated for Rev 21.0**

**by  
Glenn Morrow, Kim Seward,  
and  
Debra Spencer**

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 21.0 (Rev. 21.0).

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

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

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

Copyright © 1987 by Prime Computer, Inc. All rights reserved.

PRIME and PRIMOS are registered trademarks of Prime Computer, Inc. The Prime logo, DISCOVER, INFO/BASIC, INFORM, MIDAS, MIDASPLUS, PERFORM, Prime INFORMATION, PRIME/SNA, PRIMELINK, PRIMENET, PRIMEWAY, PRIMIX, PRISAM, PST 100, PT25, PT45, PT65, PT200, PW150, PW200, RINGNET, SIMPLE, 50 Series, 400, 750, 850, 2250, 2350, 2450, 2550, 2650, 2655, 9650, 9655, 9750, 9755, 9950, and 9955 are trademarks of Prime Computer, Inc.

#### PRINTING HISTORY

First Edition (PDR3621) March 1979 for Revision 16.3  
Second Edition (PDR3621) January 1980 for Revision 17.2  
Update 1 (PTU2600-078) December 1980 for Revision 18.1  
Third Edition (DOC3621-190) July 1982 for Revision 19.0  
First Edition  
    Volume I (DOC10080-1LA) August 1986 for Revision 20.2  
    Volume II (DOC10081-1LA) August 1986 for Revision 20.2  
    Volume III (DOC10082-1LA) August 1986 for Revision 20.2  
    Volume IV (DOC10083-1LA) August 1986 for Revision 20.2  
Update 1  
    Volume II (UPD10081-11A) July 1987 for Revision 21.0  
    Volume III (UPD10082-11A) July 1987 for Revision 21.0  
    Volume IV (UPD10083-11A) July 1987 for Revision 21.0  
Second Edition  
    Volume I (DOC10080-2LA) July 1987 for Revision 21.0

#### CREDITS

Project Support: David Brooks, Camilla Haase, Joan Karp, Alice Landy,  
Margaret Taft  
Editorial: Thelma Henner, Michael McNulty, Mary Skousgaard  
Illustration: Marjorie Clark, Mingling Chang, Susan Windheim  
Document Preparation: Julie Cyphers, Celeste Henry, Kathy Normington  
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 Friday,  
8:30 a.m. to 5:00 p.m. (EST).

### International

Contact your local Prime  
subsidiary or distributor.

## CUSTOMER SUPPORT

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

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

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



# Contents

ABOUT THIS BOOK	vii
1 OVERVIEW OF SUBROUTINES	
Functions and Subroutines	1-1
Subroutine Descriptions	1-2
Subroutine Usage	1-2
Subroutine Declarations	1-4
Subroutine Calls	1-4
Function Declarations	1-5
Function Calls	1-5
Functions Without Parameters	1-5
Subroutine Parameters	1-6
Parameter and Returned-Value	
Data Types	1-7
Optional Parameters	1-9
Optional Returned Values	1-10
How to Set Bits in Arguments	1-11
Key Names as Arguments	1-12
Standard Error Codes	1-13
Libraries and Addressing Modes	1-14
Loading and Linking Information	1-14
Satisfying the References at	
Load Time	1-14
Getting the Subroutines at	
Run Time	1-15
2 ACCESS CONTROL	2-1
3 ATTACHING	3-1
4 FILE AND DIRECTORY MANIPULATION	4-1
5 EPF MANAGEMENT	5-1
6 COMMAND ENVIRONMENT	6-1
7 SEARCH RULE SUBROUTINES	7-1

## APPENDIXES

A	OBSOLETE FILE SYSTEM SUBROUTINES	A-1
B	DATA TYPE EQUIVALENTS	B-1
C	ARGUMENT PARSING BY THE CL\$PIX SUBROUTINE	
	Overview	C-1
	CL\$PIX Operating Modes	C-1
	The Picture in Normal Mode	C-2
	The Picture in CPL Mode	C-10
	Example for CL\$PIX	C-11
	Calls Made by CL\$PIX	C-13
	INDEX OF SUBROUTINES	SX-1
	INDEX	X-1

# About This Book

The Subroutines Reference Guide is organized to give a systematic description of subroutine libraries -- sets of routines, all broadly dealing with the same subject, grouped together in one binary file. The subroutines in these libraries free the programmer from the need to rewrite the typically repeated piece of code. The programmer can, of course, make personalized subroutines as well, but will find an abundance of them already on call.

## OVERVIEW OF THIS SERIES

The Subroutines Reference Guide consists of a series of four volumes. A brief summary of the contents of each volume is given below.

### Volume I

Volume I is an introduction to the entire Subroutines Reference Guide. It describes the nature and functions of Prime's standard subroutines and subroutine libraries. It explains how subroutines can be called from programs written in Prime's programming languages: C, COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, PL/I, BASIC V/M, and PMA.

## Volume II

Volume II describes several functional groups of subroutines, dealing 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. Three chapters are devoted to subroutines related to the file system, one chapter describes system search rules, and one chapter each is devoted to those subroutines related to EPF management and to the command environment.

## Volume III

Volume III describes system subroutines. The subroutines covered in this volume are the general system calls to the operating system and standard system library. This excludes file and EPF manipulation, which are described in Volume II. Volume III also includes System Information and Metering (SIM) routines.

## Volume IV

Volume IV presents several mature libraries: the Input/Output Control System (IOCS) libraries and other I/O-related subroutines, the Application libraries, the SORT libraries, and MATHLB.

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.

## SPECIFICS OF THIS VOLUME

This volume of the Subroutines Reference Guide series presents detailed descriptions of system search rule subroutines and subroutines used in manipulating file system entities; subroutines related to EPF manipulation and the command environment are also described.

The file system subroutines (Chapters 2, 3, and 4) are divided into three groups: subroutines used in controlling access to objects, in attaching to file directories, and in operating on (creating, using, and deleting) the objects themselves.

Another subroutine group (Chapter 5) deals with the operations necessary to the initialization, execution, and maintenance of executable program format (EPF) files, and the management of dynamic storage space required for their execution.

A group of subroutines (Chapter 6) is described that enables 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.

Finally, 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.

#### SUGGESTED REFERENCES

The Prime User's Guide (DOC4130-4LA) 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.

The Programmer's Guide to BIND and EPFs (DOC8691-1LA) shows application programmers how to use the executable program format environment.

The Advanced Programmer's Guide, the companion to the Subroutines Reference Guide series, consists of four volumes:

Advanced Programmer's Guide, Volume 0: Introduction and Error Codes  
(DOC10066-1LA)

Advanced Programmer's Guide, Volume I: BIND and EPFs  
(DOC10055-1LA)

Advanced Programmer's Guide, Volume II: File System  
(DOC10056-2LA)

Advanced Programmer's Guide, Volume III: Command Environment  
(DOC10057-1LA)

These volumes provide strategies for the use of subroutines by system programmers and application programmers. In addition to explanations for each error code message, the manual provides the most complete information on the use of EPFs, of file system subroutines, and of command environments.

The following related Prime publications are also available:

Operator's Guide to System Commands (DOC9304-3LA)

System Administrator's Guide, Volume I: System Configuration  
(DOC10131-1LA)

System Administrator's Guide, Volume II: Communication Lines and  
Controllers (DOC10132-1LA)

System Administrator's Guide, Volume III: System Access and  
Security (DOC10133-1LA)

System Architecture Reference Guide (DOC9473-2LA)

#### PRIME DOCUMENTATION CONVENTIONS

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

<u>Convention</u>	<u>Explanation</u>	<u>Example</u>
UPPERCASE	In subroutine descriptions, words in uppercase indicate actual names of commands, options, statements, data types, and keywords.	FIXED BIN
lowercase	In subroutine descriptions, words in lowercase indicate variables for which you must substitute a suitable value.	key, filename
Parentheses ( )	In call statements, parentheses must be entered exactly as shown.	CALL TIMDAT(array, n)

Changes made to these pages since the last printing are identified by vertical bars in the margins. Each new routine in this package is marked with a bar beside the routine name, at the descriptions's heading.

## Overview of Subroutines

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 in this volume. 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 description of a subroutine contains the following sections (see Figure 1-1):

- Purpose. A brief description of what the subroutine does.
- 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.



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-hand 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.

Parameter 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( <u>n</u> )	Also specified as CHARACTER( <u>n</u> ), CHARACTER( <u>n</u> ) NONVARYING. Specifies a character string or array of length <u>n</u> . A CHAR( <u>n</u> ) 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( <u>n</u> ) VAR	Also CHARACTER( <u>n</u> ) VARYING. Specifies a character string or array whose length can be a maximum of <u>n</u> characters. The first 2 bytes (one halfword) of storage for a CHAR( <u>n</u> ) 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 2 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.
( <u>n</u> ) FIXED BIN	An integer array of <u>n</u> 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( <u>n</u> )	Specifies a bit string of length <u>n</u> . BIT( <u>n</u> ) 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 halfword-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 ten-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 to 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, CLIN\$,

described in Volume III, can be called with one, two or three arguments:

```
CALL CLIN$ (char);
CALL CLIN$ (char, echo_flag);
CALL CLIN$ (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 in the Subroutines Reference Guide 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.

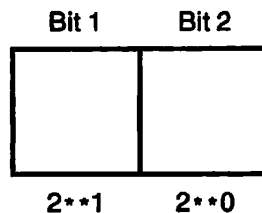
The Usage section of the subroutine descriptions gives both the function invocation and the subroutine invocation for subroutines that are likely to be called in both ways.

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.



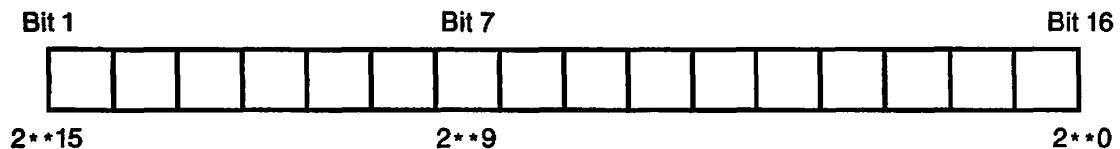
Values of Bit Positions -- Two Bits  
Figure 1-2

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.



Values of Bits in a 16-bit Argument  
Figure 1-3

#### 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 UFD named SYSCOM. 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 UFD named SYSCOM. 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 Volume 0 of the Advanced Programmer's Guide.

### Libraries and Addressing Modes

The Subroutines Reference Guide is organized to give a systematic description of subroutine libraries -- sets of routines, all broadly dealing with the same subject, grouped together 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. 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 (Prime-supplied V-mode libraries 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 Prime-supplied binary library 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 Run Time

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 Prime-supplied EPF libraries, 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 Volume I of the Subroutines Reference Guide.

## 2 Access Control

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 (UFDs), 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 Prime User's Guide and in the Advanced Programmer's Guide.

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 sub-UFD of the current UFD.

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

## Purpose

Add 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 is 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 Prime 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

## Purpose

Modify 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 is 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 Prime 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

## Purpose

Set 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 Prime 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

## Purpose

Set 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 Prime 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\$LIST

## Purpose

Obtain the contents of an object's ACL.

## Usage

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

```
CALL AC$LIST (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 | Specific ACL.                            |
| 1 | Access category.                         |
| 2 | Default access provided by specific ACL. |

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,
      2 entry_count fixed bin,
      2 entries(entry_count)char(80) var;
```

Each entry in entries is a string of the form <user-id:access>. A valid entry might be HOLMES:LUR. The user-id part 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 Prime 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

## Purpose

Convert 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:

- 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 Prime 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

## Purpose

Set 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:

- |         |   |
|---------|---|
| 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\$LIST 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 is 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\$

## Purpose

Determine 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

## Purpose

Delete 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\$

## Purpose

Obtain 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\$\$

## Purpose

Obtain the passwords of a sub-UFD of the current UFD.

## Usage

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

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

## Parameters

ufdnam

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

namlen

INPUT. Length in characters (1-32) of ufdnam.

opass

OUTPUT. Owner password for ufdnam.

npass

OUTPUT. Nonowner password for ufdnam.

code

OUTPUT. Standard error code.

## Discussion

GPAS\$\$ searches for ufdnam in the current UFD; therefore, only a simple objectname can be specified in ufdnam.

GPAS\$\$ requires Protect access to the current UFD.

The following example reads both passwords of SUBUFD:

```
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\$

## Purpose

Determine 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 Prime 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

## Purpose

Remove 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 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

## Purpose

Obtain 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 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 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

## Purpose

Set 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 for more information on priority access and how to use it, and to the Prime 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\$\$

### Purpose

Set 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 UFD. Passwords intended to be typed from the terminal should not start with a number, nor should they contain blanks or the characters = + ! , @ { } [ ] ( ) ^ < or >. Passwords should not contain lowercase characters, but 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 UFD is set to spaces. If the nonowner password supplied in the call is null, the nonowner password on the UFD 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.

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.

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

# AT\$

## Purpose

Set 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. ( $\emptyset$ )

K\$SETH Set home and current attach points. (1)

### 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\$FNTE Some part of the pathname does not exist.

E\$NRIT Use rights were unavailable at some level.

E\$NINF Some node in the tree could not be accessed, and that node's parent was missing List access.

E\$NATT A relative attach was attempted, but the current attach point was invalid.

### Discussion

AT\$ allows the user to do a pathname attach in one call. The PRIMOS pathname standard is followed:

- A partition name of <\*> means that the attach is to the current partition's MFD (the MFD containing the home directory in effect at the time of the AT\$ call).
- A pathname beginning with a partition name between angle brackets <> is a full pathname, and contains all the elements leading to the desired directory.
- A pathname beginning with an \* (asterisk) means that the attach is made relative to the home attach point in effect at the time of the AT\$ call.
- A simple objectname indicates a top-level directory, that is, a directory immediately subordinate to a partition's MFD.
- A pathname beginning with an objectname is interpreted as an absolute pathname, its first element being a top-level directory.
- 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.

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: No special action.

## AT\$ABS

### Purpose

Set 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:

0            Set only current attach point.

K\$SETH      Set current and home attach points.

#### part\_name

INPUT. Name of the disk partition on which the directory is to be found.

#### 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.

### 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.

#### 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.

## AT\$ANY

### Purpose

Set 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.

#### 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.

### Discussion

AT\$ANY differs from the AT\$ABS call in that AT\$ANY searches for the named top-level directory in all active partitions in the calling system's logical disk list, rather than on a specified partition. Partitions are searched in logical disk number order, which means that the local partitions are searched before the remote partitions.

The search begins with the first partition in the list (logical disk 0). It ends (and is considered successful) upon finding the first occurrence of the named top-level directory. Thus, if dir\_name exists on more than one partition, the second and subsequent instances of that

directory will never be found using the AT\$ANY call; to attach to such directories, use the AT\$ABS call specifying a partition name or the AT\$LDEV call specifying a logical disk number.

#### 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.

# AT\$HOM

## Purpose

Set the attach point to the home directory.

## Usage

DCL AT\$HOM ENTRY (FIXED BIN);

CALL AT\$HOM (code);

## Parameters

code

OUTPUT. Standard error code.

## 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: No special action.

# AT\$LDEV

## Purpose

Set 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:

- 0            Set only current attach point.
- K\$SETH     Set current and home attach points.

### 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\$ABS 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. (The logical disk list can be displayed 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: No special action.



# AT\$OR

## Purpose

Set 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:

0 Set only current attach point.

K\$SETH Set current and home attach points.

### code

OUTPUT. Standard error code.

## 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 the course of 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: No special action.

# AT\$REL

## Purpose

Set 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: No special action.

# ATCH\$\$

## Purpose

Set the attach point to a specified UFD and, optionally, make it the home UFD.

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.

## 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
- Closing file system objects by name or file unit number
- Manipulating names, suffixes, attributes, read/write modes, and directory quotas

The subroutines described in this chapter do not use the search rules facility. Chapter 7 of this volume includes descriptions of OPSR\$ and OPSRS\$, which use search rules to locate and open files.

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

AP\$FX\$	Append a specified suffix to a pathname.
CH\$MOD	Change the open mode of an open file.
CL\$FNR	Close a file by name and return a bit string indicating 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 a directory open on a file unit.
DIR\$RD	Read sequentially the entries of a directory open on a file unit.
DIR\$SE	Return entries meeting caller-specified selection criteria in a directory open on a file unit.
ENT\$RD	Return the contents of a named entry in a directory open on a file unit.
EQUAL\$	Generate a filename based on another name.
EXTR\$A	Return a 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 the pathname of a 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.
LUFSK\$	List the disks a given user is using.
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 usage 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.
UNIT\$S	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.



## APSF\$

### Purpose

Append a specified suffix to a pathname.

### Usage

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

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

### Parameters

in\_pathname

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

out\_pathname

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

code

OUTPUT. Standard error code. Possible values are:

-1        Suffix already present, pathname remained unchanged.

0        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 Prime 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.

APSF\$ 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.

APSF\$ is often helpful because SRSFX\$ returns two parts to a name -- the basename 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

### Purpose

Change 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

### Purpose

Close 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 UNIT\$ 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

## Purpose

Close 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

### Purpose

Close 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\$\$

## Purpose

Change 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\$\$

### Purpose

Create 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\$

### Purpose

Create 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

## Purpose

Create 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

## Purpose

Search 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 Prime 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 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. 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, and 31 halfwords from Rev. 20.0 onward.

**entryname**

Name of the entry, in uppercase.

**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 non-owner 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.

**file\_info.special**

If set, the file is special (e.g., DSKRAT, BOOT, MFD) and cannot be deleted.

**file\_info.rwlock**

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

`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.

`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).

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

### Purpose

Read 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.	- Returns NO entry
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. *Halfwords*

code

OUTPUT. Standard error code.

Discussion

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  name CHAR(32),
      2  pw_protection BIT(16) ALIGNED, 2
      2  non_default_protection BIT(1) ALIGNED, 2.
      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.

ecw.length

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

**name**

The name of the entry, in uppercase, left-justified and filled with spaces.

**pw\_protection**

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\_protection**

Set to true ('1'b) if the entry is not default-protected; it is either protected specifically or by an access category.

**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.

**file\_info.special**

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

`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.

`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.

`dtc`

Date and time the file was last created.

`dta`

Date and time the file was last accessed.



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

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

Note

Calls to DIR\$RD and ENT\$RD should not be made on the same directory file unit unless DIR\$RD is called with the K\$INIT key following each ENT\$RD call.

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

## Purpose

Return entries meeting caller-specified selection criteria in a directory open on a file unit.

## Usage

```
DCL DIR$SE ENTRY (FIXED BIN, FIXED BIN, BIT(1), PTR, PTR,
                  FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN,
                  FIXED BIN, FIXED BIN);
```

```
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.

**return\_ptr**

INPUT -> OUTPUT. Pointer to caller's return structure for selected entry data. This parameter points to where the subroutine places the output. See the Discussion section.

**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 zero.

**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 4-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-count 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 4).
E\$EOF	There are no more entries in the directory to be selected.
E\$ST19	Selection criteria involving RBF file 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.

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 Prime 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.

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, and 31 halfwords from Rev. 20.0 onward.

## entryname

Name of the entry, in uppercase.

## protection.owner\_rights

Rights granted to a user when attached to the containing directory with owner rights.

## 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.

## protection.non\_owner\_rights

Rights granted to a user when attached to the containing directory with nonowner rights.

## file\_info.long\_rat\_hdr

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

## 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.

`file_info.special`

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

`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.



`non_default_acl`

This bit is set if the object is not protected by the default ACL -- that is, if it is protected by a specific ACL or by an access category.

`logical_type`

This is an additional file type to the physical file type described in file\_info.type. Possible values are:

0            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.

`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.

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

## Purpose

Return 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\$

## Purpose

Generate 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 object name and a generation pattern. The latter contains "commands" that specify how to transform the object name into a new name called the generated name. This routine performs that transformation. Name generation is discussed in the Prime 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

## Purpose

Return 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.

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

## Purpose

Delete 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\$

## Purpose

Return 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,
  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 spare3 BIT (1),
    3 attach_point BIT (1),
    3 write BIT (1),
    3 read BIT (1),
  2 file_type FIXED BIN,
  2 rwlock FIXED BIN,
  2 position FIXED BIN (31),
  2 system_name CHAR(32) VAR,
  2 ldevno FIXED BIN,
  2 packname CHAR(32) VAR;
```

#### 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\$

## Purpose

Verify 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 entry names).

### 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

## Purpose

Force 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 INPUT. Standard return code that is E\$DISK when a disk error occurred on the file referenced by funit. If code is not supplied as an argument, then disk errors are not reported.

## 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, since the system automatically updates all changed file system information to the disk at least once per minute. Under PRIMOS II, the FORCEW routine has no effect.

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

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\$

## Purpose

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, buflen, 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 varying 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 full pathname. The buffer contains no valid data.

E\$FNTE    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.

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 define what information the examples (in uppercase) actually represent.

<disk\_name>MFD  
<SPOOLD>MFD

<disk\_name>ufd\_name  
<SPOOLD>SPOOLQ

<disk\_name>ufd\_name1>ufd\_name2>file\_name  
<SALESD>WEST.COAST>YTD.1979>MARCH

<disk\_name>ufd\_name>segment\_directory\_name  
<OPSYST>PR4.64>VPRMOS

<disk\_name>ufd\_name>segment\_directory\_name>entry\_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\$

### Purpose

Determine 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\$

### Purpose

Return information on the system's list of logical disks.

### 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. 20.2, no more than 62 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, physical device number, system name if remote, priority ACL status, and write-protect status.

The structure returned by LDISK\$ has the following format:

```
DCL 1 disk_list,
    2 version FIXED BIN,
    2 info_count FIXED BIN,
    2 info(info_count),
    3 priority_acl BIT(1),
    3 write_protected BIT(1),
    3 rsvd BIT(14),
    3 ldevno FIXED BIN,
    3 pdevno FIXED BIN,
    3 disk_name CHAR(32) VAR,
    3 system_name CHAR(32) VAR;
```

## version

Caller-supplied version number of the structure. Must currently be 1.

## info\_count

Number of entries returned in the info array (described next). Info\_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 62.

`info.priority_acl`

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

`info.write_protected`

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

`info.ldevno`

Logical device number of the partition.

`info.pdevno`

Physical device number of the partition.

`info.disk_name`

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

`info.system_name`

Name of the system on which the disk is physically added. Null for local disks. Currently, this name is one to six characters long, but space for 32 is reserved.

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\$

## Purpose

List 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. Maximum that the subroutine can return at Rev. 20.2 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.



The structure pointed to by return\_ptr has the following format:

```
DCL 1 rtn_struct 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: 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 one to six characters long, but space for 32 characters is reserved.

info.ldev

Logical disk number associated with this partition.

info.system\_name

If the disk in this entry is remote, the system name to which it is physically attached. Currently, system names are one to six characters long, but space for 32 characters is reserved.

#### 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.

# PAR\$RV

## Purpose

Return 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 one to six 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 Prime 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\$\$

## Purpose

Read, write, position, or truncate a file.

## Usage

DCL PRWF\$\$ ENTRY (FIXED BIN, FIXED BIN, PTR OPTIONS (SHORT), FIXED BIN,  
FIXED BIN (31), FIXED BIN, FIXED BIN);

CALL PRWF\$\$ (rwkey+poskey+modekey, funit, loc(buf), nhw, pos,  
rnhw, code);

## Parameters

### 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 the following:

- 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 to 15 for PRIMOS II, 1 to 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. 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 code E\$EOF will be returned even if some data was transferred.

### Discussion

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 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 buffer-full 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. 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).

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 ten 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 ten 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 ten 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 and DAM files
*/

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

#include 'syscom>keys.pl1';
#include 'syscom>errrd.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  errpr$        entry(bin, bin, char(*), bin, char(*), bin);
dcl  user_proc      entry;
dcl  prwf$$        entry (fixed binary(15),
                          /* keys (rwkey+poskey+mode) */
                          fixed binary(15), /* unit to perform action on */
                          pointer options(short),
                          /* address of data buffer */
                          fixed binary(15), /* words to read or write */
                          fixed binary(31), /* position value */
                          fixed binary(15), /* actual words read or 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$dkfl
    then do;
      call errpr$(k$irtn, code, '', 0, 'cp$$fl', 6);
      call user_proc;          /* Wait for response */
      go to a;
    end;
  else do;
    err_info = 2;
    return;
  end;
end;
return;
end cp$$fl;
/*****/

```

More examples of the use of PRWF\$\$ Are given with the file-system examples in Volume I.

#### 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

### Purpose

Return directory quota and disk record usage 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 6 for Rev. 20.2.

type

OUTPUT. Type of directory (input):

0	Quota Directory
1	Non-quota Directory

code

OUTPUT. Standard error code:

E\$NINF Insufficient access to read quota.

Discussion

Quota and disk usage accounting concepts are explained in the System Administrator's Guide.

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 that rec\_time\_product was last updated, in standard File-system Date Format (see Appendix C of Volume III for more information about this format).

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

The accounting meter is a counter that acts as an unsigned 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:

$$\text{TIME} = (\text{Current date/time}) - (\text{Date/time quota last modified})$$
$$\text{USAGE} = \text{USAGE} + (\text{quota\_used}) * \text{TIME}$$

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

### Purpose

Set 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 non-quota 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 whenever 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\$\$

### Purpose

Position in or read 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\$

### Purpose

Read a line of characters from an ASCII disk file.

### Usage

DCL RDLIN\$ ENTRY (FIXED BIN, CHAR(\*), FIXED BIN, FIXED BIN);

CALL 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.

### 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.

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\$\$

### Purpose

Set or modify 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 <u>attributes</u> array. The second halfword of <u>attributes</u> 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 <u>attributes</u> .
K\$DTB	Set date/time backed up from both halfwords of <u>attributes</u> .
K\$DTC	Set date/time created from both halfwords of <u>attributes</u> .
K\$DTA	Set date/time last accessed from both halfwords of <u>attributes</u> .
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 .

Users should use the K\$DMPB key with care, since 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\$STRUN 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:

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	<u>namlen</u> 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.
E\$WTPR	The disk is write-protected.
E\$NINF	An error occurred during search of the directory, and List access was not available.
E\$FNTE	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 of a 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 system-wide 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:

<u>Value</u>	<u>Bits 5,6</u>	<u>Meaning</u>
0	0,0	Use system-wide RWLOCK to regulate concurrent access.
1	0,1	Allow arbitrary readers or one writer.
2	1,0	Allow arbitrary readers and one writer.
3	1,1	Allow arbitrary readers and arbitrary writers.

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: DIR\$LS, DIR\$RD, 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 usage 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 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.

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

### Purpose

Delete 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

### Purpose

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

### Purpose

Open 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 (SEGDM)      |
| 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:

- |         |   |
|---------|---|
| K\$NSAM | Create a sequential access file.              |
| K\$NDAM | Create a direct access file.                  |
| K\$NSGS | Create a sequential access segment directory. |
| K\$NSGD | Create a direct access segment directory.     |
| K\$NCAM | Create a contiguous access 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.

### 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\$\$

### Purpose

Position in, read an entry in, or modify the size of a segment directory.

### Usage

DCL SGDR\$\$ ENTRY (FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN,  
FIXED BIN);

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

### Parameters

key

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:

The position given by entrya if this position is full.

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.

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.pll';
#include 'syscom>errd.pll';

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 errpr$     entry(bin, bin, char(*), bin, char(*), bin);
dcl srch$$     entry(bin, bin, bin, bin, bin, bin);
dcl cp$$fl     entry(bin, bin, bin, bin);
                /* cp$$fl is defined in example 6 for PRWF */
dcl sgdr$$     entry (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$spos, tunit, entry_no, entryb, code);
    if code ^= 0
        then go to err_rtn_2;
    if entryb < 0
        then do;
            call errpr$ (k$irtn, e$null, 'Unrecoverable
                error', 19, 'cp$$sd', 5);
            stop;
        end;

    if rtnval = 1
        then do;

```

```

/* found a nonnull entry in source, */
/*      open it and same entry in target*/

      call srch$$ (k$read + k$iseg + k$getu, sunit, 0,
                  sfunit, type, code);
      if code ^= 0
        then go to err_rtn_1;

      newfil = k$nsam;
      if type = 1
        then newfil = k$ndam;
      if type = 2
        then newfil = k$nsogs;
      if type = 3
        then newfil = k$nsogd;
      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\$

### Purpose

Return 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\$\$

## Purpose

Open, close, delete, change access, or verify the existence of an object.

## Usage

DCL SRCH\$\$ ENTRY (FIXED BIN, CHAR (32) ~~VAR~~, 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 run file.

### 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, 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 below for use of this key.

#### 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 (This is 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). K\$CURR can 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 to 126 (1 to 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.

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.

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\$\$, RDN\$\$, 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 the 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.

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.

Operations on Subdirectories: The contents of entries of subdirectories can be read through calls to ENT\$RD, DIR\$LS, DIR\$RD, DIR\$SE, 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, perform the following 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 named '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$nsgrs;
            else if type = dam_segdir
                then newfil = k$nsgrd;

```

```

    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\$

## Purpose

Search 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+ref+newfile, object_name, funit, type, n_suffixes,
             suffix_list, basename, suffix_used, code);
```

```
chrpos = SRSFX$ (action+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 <u>object_name</u> for reading on <u>funit</u> .
K\$WRIT	Open <u>object_name</u> for writing on <u>funit</u> .
K\$RDWR	Open <u>object_name</u> for reading and writing on <u>funit</u> .
K\$CLOS	Close <u>object_name</u> .
K\$DELE	Delete <u>object_name</u> .
K\$EXST	Check on existence of <u>object_name</u> .
K\$VMR	Open <u>object_name</u> for VMFA read. Valid only if <u>object_name</u> is an EPF-format run file.

ref

INPUT. Modifies the action key as follows:

K\$IUFD	Search for <u>object_name</u> 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, 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 (This is 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 (remains unchanged). A pathname of '' (null string) opens 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.

#### 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 routine returns when a filename suffix is found or the suffix list is exhausted.

If a file is found, the index (in the suffix list) of the last suffix in the filename is returned; if no file is found, or if none of the suffixes in the list is on the found filename, an index of 0 is returned.

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.

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.pll
$Insert syscom>errd.ins.pll

/* 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),
    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);

/* 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$get (arg_line, 150, code);
if code ^= 0
    then call errpr$(k$nrtn, 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(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 errpr$(k$irtn, status, addr (pathname) ->
                    bvs.chars, length (pathname), '', 0);
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;
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\$

### Purpose

Verify a supplied string as 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 Prime 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.

### Discussion

Pathnames are discussed in Prime User's Guide.

The TNCHK\$ call does not check on the existence of the object represented by pathname, but only that the pathname obeys the rules for constructing pathnames. 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\$\$

### Purpose

Open 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\$

## Purpose

Return 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\$

### Purpose

Return 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 Prime 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\$

### Purpose

Write 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.

### 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.

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.

## 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; 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 EPF'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.

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\$MAP Map the procedure images of an EPF file into virtual memory.

EPF\$RUN Combine functions of EPF\$ALLC, EPF\$MAP, EPF\$INIT, and EPF\$INVK.

REMEPF\$ Remove an EPF from a user's address space.

RPL\$ Replace one EPF runfile with another.



# EPF\$ALLC

EPF\$AL is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Perform 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 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 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Return 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, 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 values are:

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 invokable features are returned is shown below. Refer to the Advanced Programmer's Guide for explanations of each bit.

```
1 epf_info based,
  2 command_flags,
    3 wildcards bit(1),
    3 treewalks bit(1),
    3 iteration bit(1),
    3 verify bit(1),
    3 file_types,
      4 file bit(1),
      4 directory bit(1),
      4 segdir bit(1),
      4 acat bit(1),
      4 rbf bit(1),
      4 reserved bit(7),
  2 name_generation_position fixed bin(15);
```

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 is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Deactivate 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 data base 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 is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Perform 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 re-initialization 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\$BLTE	An invalid EPF LTE linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.
E\$BLTD	An invalid EPF LTD linkage descriptor type has been found within the EPF file. Resubmit the file to BIND.
E\$BPAR	An undefined epf_id has been passed as a parameter, probably indicating that the EPF was not successfully mapped into memory by EPF\$MAP.
E\$BVER	An invalid EPF version was detected. Resubmit the file to BIND.
E\$EPFT	An invalid EPF type field was detected when trying to allocate storage. Resubmit the file to BIND.

### 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 is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Initiate 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),
  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 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 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 EPF\$MAP, EPF\$ALLC, EPF\$INIT, and EPF\$INVK.

The address of the starting entry control block for the EPF is found from the Control Information Block (CIB) within the EPF, and the EPF is invoked by issuing a PCL instruction to the ECB.

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.

```

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 above 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\_struct pointer is:

```

1 rtn_fcn_struct,
  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 de-allocates 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\$MAP

EPF\$MP is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Map 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: EPF\$MAP, EPF\$ALLC, EPF\$INIT, EPF\$INVK. Refer to the Advanced Programmer's Guide 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:

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 re-configured 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.

In response to any of these three messages, the user can release temporary segments in these ways:

- 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)

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\$RUN

EPF\$RN is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Combine functions of EPF\$MAP, EPF\$ALLC, 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 EPF\$MAP, EPF\$ALLC, 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 routine performs all the appropriate calls to execute an EPF file. It maps and allocates the linkage and static data areas, initializes them, invokes the EPF, and optionally returns the EPF memory resources to the system free pool. The EPF file must first be opened for a VMFA-read; that is, you first must call either SRCH\$\$ 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 for a full discussion on calling 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.

## REMEPF\$

### Purpose

Remove 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

Forcibly remove if process-class library EPF is initialized.

K\$NO\_FRC\_DEL

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<sub>n</sub> 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<sub>n</sub> 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\$

### Purpose

Replace 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 .RPn file if it is currently in use; otherwise, a null string.

no\_query

INPUT. If this bit is set, no query for changing the file name will prompt the user, and no messages are displayed. If it is unspecified by the user, the routine defaults to query displays.

code

OUTPUT. Standard error code. Possible values are:

-1	Returned as a warning if at least one RPn 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 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<sub>n</sub>, where <sub>n</sub> is a decimal integer from 0 through 9. RPL\$ replaces the old EPF file with this new .RUN file, but the .RP<sub>n</sub> 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 Volume III of the Advanced Programmer's Guide.



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	Returns caller's current command environment depth.

# CE\$BRD

## Purpose

Return caller's maximum command environment breadth.

## Usage

DCL CE\$BRD ENTRY () RETURNS (FIXED BIN(15));

max\_ce\_brth = CE\$BRD();

## Parameters

max\_ce\_brth

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

### Purpose

Return 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

## Purpose

Parse 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:

```
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 pll_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 pll\_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 pll\_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, 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 type UNCL has been given more than once or has been given for an option argument parameter.

#### 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\$

## Purpose

Invoke a command from a running program.

## Usage

```
DCL CP$ ENTRY (CHAR(1024) VAR, FIXED BIN(15), FIXED BIN(15),
               1,
               2 BIT(1),
               2 BIT(1),
               2 BIT(14),
               PTR, PTR);
```

```
CALL CP$ (command_line, status, 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 fourteen 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**

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 Volume III of the Subroutines Reference Guide.

**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 Volume III of the Advanced Programmer's Guide.

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 inputs 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$('RESUME PROGS>ORDER WRENCH', 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

## Purpose

Retrieve 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\$FNTE    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 Prime 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

### Purpose

Set 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 Prime 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

### Purpose

Return 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 Prime 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.

```
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 carriage 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\$FNTE (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

### Purpose

Retrieve 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

### Purpose

Set 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Returns caller's current command environment depth.

### Usage

DCL RD\$CE\_DP ENTRY (FIXED BIN);

CALL RD\$CE\_DP (com\_env\_dpth);

### Parameters

com\_env\_dpth

OUTPUT. The current depth 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 routine returns the command level depth at which the user is currently operating.

The maximum command environment depth is set on a per-user basis by the System Administrator. The user can retrieve this maximum for comparison with the current depth by using the CE\$DPT 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 Rule Subroutines

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, Volume II. 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.

Most search rule subroutines can be invoked by either their full name or a six-character synonym name. A list of the subroutines described in this chapter follows.

<u>Routine</u>	<u>Function</u>
OPSR\$	Locates a file using a search list and opens the file. Creates a file if the file sought does not exist.
OPSR\$\$	Locates a file using a search list and a list of suffixes. Opens the located file, or creates a file if the file sought does not exist.
SR\$ABSDS	Disables 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	Adds a rule to the beginning of a search list or before a specified rule.
SR\$ADDE	Adds a rule to the end of a search list or after a specified rule.
SR\$CREAT	Creates a search list.
SR\$DEL	Deletes a search list.
SR\$DSABL	Disables 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	Enables an optional search rule. Enabled rules can be disabled using SR\$DSABL or SR\$ABSDS.
SR\$EXSTR	Determines if a search rule exists.
SR\$FR_LS	Frees list structure space allocated by SR\$LIST or SR\$READ.
SR\$INIT	Initializes all search lists to system defaults.
SR\$LIST	Returns the names of all defined search lists.
SR\$NEXTR	Reads the next rule from a search list.
SR\$READ	Reads all of the rules in a search list.
SR\$REM	Removes a search rule from a search list.
SR\$SETL	Sets the locator pointer for a search rule.
SR\$SSR	Sets 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\$

## Purpose

Locate a file using a search list and open the file. OPSR\$ 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 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 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 circumstances 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 as follows:

- |   |   |
|---|---|
| 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:

- |         |   |
|---------|---|
| 0       | 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, Volume II 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.

# OPSR\$

## Purpose

Locate and open a file using a search list and a list of suffixes. OPSR\$ 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 OPSR$ 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 OPSR$ (list_name, referencing_dir,
            valid_types, action+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, 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 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. 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 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).

### 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 argument or the 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, OPSR\$ uses the search rules facility to perform a search using the list\_name and a list of suffixes. If you supply a full pathname, OPSR\$ 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, OPSR\$ uses funit to return the file unit number assigned by PRIMOS.

### type

OUTPUT. The type of the object that OPSR\$ successfully accessed. Possible values are as follows:

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:

0	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\$ 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, OPSR\$ does not require that object\_name include the filename suffix. Instead, OPSR\$ uses a list of suffixes when searching for a file.

OPSR\$ searches each rule in the search list for the specified filename plus the first listed suffix, then the second suffix, and so on. After testing a search rule for the combination of object\_name and each suffix in turn, OPSR\$ checks for object\_name with no suffix. If unsuccessful, OPSR\$ 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), OPSR\$ appends suffixes from the list to object\_name, creating filenames with multiple suffixes, such as MYFILE.RUN.CPL. However, OPSR\$ does not append a suffix to an identical suffix (such as 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 OPSR\$; 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, OPSR\$ locates either TESTFILE.CPL, TESTFILE.F77 or TESTFILE and opens it for update.

```

/* Sample PL/I program for the OPSR$ subroutine */
OPEN_PROG: PROCEDURE;
%INCLUDE 'SYSCOM>KEYS.PL1';
/* Declarations */
DCL OPSR$ 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 OPSR$(LIST, REF, K$FILE, K$RDWR, OBJNAME,
            FUNIT, TYPE, N_SUFFIX, ADDR(SUFFIX_LIST), BASENAME,
            SUFFIX_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 OPSR\$ 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_SUFFIX
INTEGER*4 SUFFIX_PTR
INTEGER*2 BSIZE, BPLUS(32)
CHARACTER*32 BASENAME
INTEGER*2 SUFFIX_USED
CHARACTER*128 FOUND
INTEGER*2 CODE

```

C Declarations of suffix list

```

INTEGER*2 SUFFIX_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, SUFFIX_LIST(1))
EQUIVALENCE (SUF1, SUFFIX_LIST(2))
EQUIVALENCE (LSUF2, SUFFIX_LIST(18))
EQUIVALENCE (SUF2, SUFFIX_LIST(19))

```

```

C  Assignments
    LIST(1:6) = 'MYLIST'
    LSIZE = 6
    REF(1:1) = ''
    REFSIZE = 1
    OBJNAME(1:8) = 'TESTFILE'
    OBJSIZE = 8
    FUNIT = 3
    N_SUFEX = 2
    SUFX_PTR = LOC(SUFEX_LIST(1))
    FOUND = ''
    SUF1(1:4) = '.CPL'
    LSUF1 = 4
    SUF2(1:4) = '.F77'
    LSUF2 = 4
C  Subroutine call
    CALL OPSRSS(LPLUS, RPLUS, K$FILE, K$RDWR,
*   OBJPLUS, FUNIT, TYPE, N_SUFEX, 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 is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Disable an optional search rule in a specified search list. SR\$ABSDS 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:

0            Operation succeeded. SR\$ABSDS returns 0 even if the rule was not enabled.

E\$LIST      Search list does not exist.

E\$RULE      Search rule cannot be located. Rule may be non-existent 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Add a search rule before an existing search rule in a search list. SR\$ADDB 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 exactly 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:

0	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 <u>old_rule</u> 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Add a search rule to the end of a search list or add 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 exactly 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:

0	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 <u>old_rule</u> 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, use SR\$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 the search 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Create 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:

0            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

### Purpose

Delete 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:

0            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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Disable 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:

0            Operation succeeded. This return code does not indicate whether or not the rule is actually disabled. SR\$DSABL returns 0 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 non-existent 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Enable 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:

0	Operation succeeded.
E\$LIST	Search list does not exist.
E\$RULE	Search rule cannot be located. Rule may be non-existent 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Determine 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Free 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:

0            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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Initialize 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:

- |         |  |
|---------|--|
| 0       | Operation succeeded. All search lists initialized.   |
| E\$FNTF | A system default file contains an -insert keyword that refers to a non-existent 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\$LIST is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Return 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:

0            Operation succeeded.

E\$BVER    Version number is invalid.

### Structure Description

The parameter output\_ptr points to a structure, list\_struc, of the following format:

```
DCL 1 list_struct,  
    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, 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;
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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Read 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:

- 1 K\$TEXT Rule is an ordinary text string.
- 2 K\$HMDR Rule is the [home\_dir] keyword.
- 3 K\$ORDR Rule is the [origin\_dir] keyword.
- 4 K\$RFDR Rule is the [referencing\_dir] keyword.
- 8 K\$KEYW 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:

- 0 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 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 is an alternate name, which is required for FTN and is optional for other languages.

## Purpose

Read 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:

0	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\_struct, of the following format:

```
DCL 1  rules_struct,
      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, Volume

II. 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

## Purpose

Remove 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:

0	Operation succeeded.
E\$LIST	Search list does not exist.
E\$RULE	Rule cannot be found in search list specified. Rule may be non-existent 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 exactly 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 is an alternate name, which is required for FTN and is optional for other languages.

### Purpose

Modify 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:

0            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$SETL EXTERNAL ENTRY (FIXED BIN(31), PTR, FIXED BIN);
DCL LOC2 PTR;
DCL 1 LOCATOR2 DEFINED (LOC2),
    2 FAULT2 BIT(1),
    2 RING2 BIT(2),
    2 FORMAT2 BIT(1),
    2 SEGNO2 BIT(12),
    2 WORDNO2 BIT(16);
SEGNO2 = '10011111111'b;
WORDNO2 = '01010101010101'b;
/* Perform the calls */
RETV = SR$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(RETV, LOC2, CODE);
IF (CODE = 0) THEN
    PUT SKIP LIST('Locator pointer set');
ELSE GO TO A;
RETV = 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

## Purpose

Set a search list using a user-defined search rules file. SR\$SSR 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 non-existent files. The default value for this argument is 0.

code

OUTPUT. Standard error code. Possible values are:

- 0            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\$FNTE    Either the search rules file does not exist or an -insert or -system keyword refers to a non-existent 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 indicated by 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.

# APPENDIXES



# **A**

## **Obsolete File System Subroutines**

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.

## Purpose

ATCH\$\$ attaches to a UFD and, optionally, makes it the home UFD. In attaching to a directory, the subroutine ATCH\$\$ specifies where to look for the directory. ATCH\$\$ specifies that a User File Directory (UFD) is in the Master File Directory (MFD) on a particular logical disk, in a subdirectory in the current UFD, or in the home UFD.

## Usage

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

ufdnam	The name of the UFD to be attached (integer array). If <u>key</u> is K\$IMFD and <u>ufdnam</u> is the key K\$HOME, the home UFD is attached. If the reference subkey is K\$ICUR, <u>ufdnam</u> is the name of an array that specifies the name of the UFD to attach to.
namlen	The length in characters (1-32) of <u>ufdnam</u> (INTEGER*2). <u>namlen</u> may be greater than the length of <u>ufdnam</u> provided that <u>ufdnam</u> is padded with the appropriate number of blanks. If ufdnam = K\$HOME, <u>namlen</u> is disregarded.
ldisk	The number of the logical disk to be searched for <u>ufdnam</u> when key = K\$IMFD (INTEGER*2). The parameter <u>ldisk</u> must be a logical disk that is started up. Other values for <u>ldisk</u> are:
K\$ALLD	Search all started-up <u>local</u> logical devices in logical device order (then likewise all such <u>remote</u> devices), and attach to the UFD in which <u>ufdnam</u> 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 ufdnam. passwd can be specified as 0 if attaching to the home UFD. If the reference subkey is K\$IMFD or K\$ICUR, passwd must be the name of a three-halfword array that specifies one of the passwords of ufdnam. 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 ufdnam in MFD on ldisk.

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

The SETHOME subkey, K\$SETH, may be added to the REFERENCE subkey as K\$IMFD+K\$SETH, which will set the current UFD to the home UFD after attaching. If the REFERENCE subkey is K\$ICUR, or if ufdnam is 0, ldisk is ignored, and it is usually specified as 0.

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

### Discussion

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

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

Examples

1. Attach to home UFD:

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

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

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

# 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.

## Purpose

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

## Usage

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

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

filnam	The name to be given the new UFD (input).
namlen	The length in characters (1-32) of <u>filnam</u> (16-bit integer).
owner-pw	A six-character array containing the owner password for the new UFD. If <u>owner-pw</u> (1) = 0, the owner password is set to blanks. <u>owner-pw</u> is ignored if an ACL directory is being created.
nonowner-pw	A six-character array containing the nonowner password for the new UFD. If <u>nonowner-pw</u> (1) is 0, the nonowner password is set to zeros. Any password given to ATCH\$\$ matches a nonowner password of zeros. <u>nonowner-pw</u> is ignored if an ACL directory is being created.
code	A 16-bit integer variable to be set to the return code from CREA\$\$.
	Possible values follow.
E\$BNAM	The supplied name is illegal.
E\$BPAR	The name length is illegal.
E\$EXST	An object with the given name already exists.

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

E\$WTPR    The disk is write-protected.

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

E\$NATT    The current attach point is invalid.

### Discussion

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

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

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

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

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

### Example

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

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

# 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.

## Purpose

CREPW\$ creates a new password directory.

## Usage

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

```
CALL CREPW$ (name, name-length, owner-pw, non-owner-pw, code);
```

name	Name of the directory to be created (input).
name-length	Length of the name in characters (input).
owner-pw	Password which must be used to attach with owner rights (input).
nonowner-pw	Password that must be used to attach with nonowner rights (input).
code	Standard error code (output). Possible values are:
	E\$BNAM The supplied name is illegal.
	E\$BPAR The name length is illegal.
	E\$EXST An object with the given name already exists.
	E\$NRIT Add rights were not available on the current directory.
	E\$WTPR The disk is write-protected.
	E\$NINF An error occurred, and list rights were not available on the current directory.
	E\$NATT The current attach point is invalid.

Discussion

CREPW\$ is used to create new directories. It always creates a password directory. Add access is required on the current directory.



# RDEN\$\$

## Note

In new programming, use of the DIR\$RD or ENT\$RD subroutine in place of the RDEN\$\$ subroutine is recommended. These subroutines are described in Chapter 4.

## Purpose

RDEN\$\$ positions in or reads from a UFD.

## Usage

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

**key**                    A 16-bit integer variable specifying the action to be taken. Possible values are:

**K\$READ**      Advance to the start of the first or next UFD entry and read as much of the entry as will fit into buffer. Set rnw to the number of 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 rnw 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 UFD as a 32-bit integer in filnam.

**K\$UPOS**      Set the current position in the UFD from the 32-bit integer in filnam. This key should be used only with a position of 0.

**K\$POSN**      Return access category entries.

funit	A unit on which a UFD is currently opened for reading (INTEGER*2). (A UFD may be opened with a call to SRCH\$\$.)
buffer	A one-dimensional array into which entries of the UFD are read.
buflen	The length, in halfwords, of <u>buffer</u> (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 <u>filnam</u> . This variable is only used with K\$NAME.
code	An INTEGER*2 variable to be set to the return code:  E\$FNTF The entry is not in the directory.  E\$EOF No more entries.  E\$BFTS Buffer is too small for the entry.

### Discussion

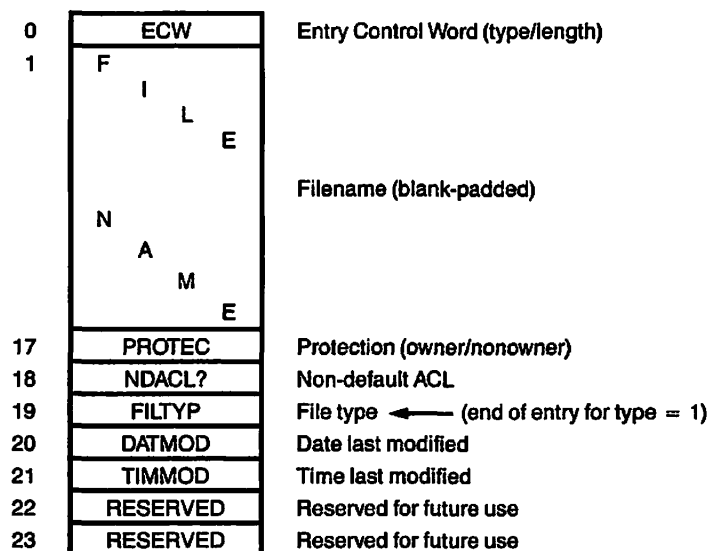
RDEN\$\$ is used to read entries from a UFD. rnhw halfwords are returned in buffer, and the file unit position is advanced to the start of the next entry.

#### Caution

Directory positioning is obsolete and should not be necessary.

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

The complete format of currently defined entries is given in Figure A-1 and discussed below for revisions before 19. (For Rev. 19 format, see DIR\$RD.) All numbers are decimal unless preceded by a colon (:).



File Entry Format  
Figure A-1

#### 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 UFD 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 III) when it modifies the file. If this bit is set, the time-date field for the file will not be current because PRIMOS II doesn't update the date/time stamp when it modifies a file.

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

bits 5-6 Setting of the read/write lock. (See below.)

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

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

### The Read/Write Lock

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

The meaning of the lock values is:

<u>Value</u>	<u>Bits 5,6</u>	<u>Meaning</u>
0	0,0	Use system-wide RWLOCK to regulate concurrent access.
1	0,1	Allow arbitrary readers or one writer.
2	1,0	Allow arbitrary readers and one writer.
3	1,1	Allow arbitrary readers and arbitrary writers.

New files are initially created with a per-file read/write lock of 0.

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

The per-file read/write lock value is read by RDEN\$\$ . It is set by a SATR\$\$ call with a key of K\$RWLK. The desired value is supplied in

bits 15 and 16 of ARRAY(1), the remaining bits of which must be 0. On old partitions, the SATR\$\$ call fails with an error code of E\$OLDP. Owner rights to the containing UFD are required, otherwise the call fails with an error code of E\$NRIT. An attempt to set the lock value of a UFD fails with an error code of E\$DIRE. If the SATR\$\$ call requests a lock value which is more restrictive than the current usage of the file, the file's lock value is changed and current users of the file are unaffected, but any new openings subsequently requested are governed by the new lock value. It is unspecified what happens when bits 1-13 of ARRAY(1) are not 0.

The commands MAGSAV and MAGRST properly save and restore the per-file read/write lock along with the file itself. Existing backup tapes without saved read/write locks on them are restored with read/write locks of 0, so the system-wide RWLOCK setting continues to control access to such files.

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

### Examples

1. Read next entry from new or old UFD:

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

2. Position to beginning of UFD:

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

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

```
/******
rd$dir:
    proc(dunit, rden_ptr, code);
dcl  dunit          bin,      /* unit directory is open on */
     rden_ptr       pointer, /* pointer to rden_buffer */
     code           bin;     /* standard error code */

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

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

```

dcl i                bin;
dcl trim            builtin;

/*****

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

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

```

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

```

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

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

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

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

*****/

```

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

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

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

call close$(unit);

rden_buffer(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.

## Purpose

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

## Usage

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

**action**            A 16-bit key indicating the action to be performed.  
Possible values are:

K\$READ    Open pathname for reading on funit.  
K\$WRIT    Open pathname for writing on funit.  
K\$RDWR    Open pathname for reading and writing  
          on funit.  
K\$DELE    Delete file pathname.  
K\$EXST    Check on existence of pathname.  
K\$CLOS    Close pathname (not funit).  
K\$GETU    Open pathname on an unused file unit  
          selected by PRIMOS. The unit number is  
          returned in funit.  
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 for type are:

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

code    A 16-bit integer variable set to the return code. If no errors, code is 0.

## 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.

The table that follows 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 Guide, Volume I 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  
Data Type Equivalents

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

Table B-1 (continued)  
Data Type Equivalents

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;	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		char NAME[n]; char *NAME;				STRING[n]	CHAR(n) VARYING
32-bit pointer		Pointer (32IX-mode)					POINTER OPTIONS (SHORT)
48-bit pointer		Pointer (64V-mode)				Pointer	POINTER

**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 Guide, Volume 1*.

The BASIC/VM column lists FTN data types to be declared in the SUB FORTRAN statement in a BASIC/VM program.

# C

## Argument Parsing by the CL\$PIX Subroutine

### 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.
char1	Arbitrary character string up to 80 bytes long, not mapped.

tree	PRIMOS pathname up to 128 bytes long, mapped to uppercase. Wildcard characters are allowed.
entry	Filename, up to 32 bytes long, mapped to uppercase. Wildcard characters are allowed.
id	PRIMOS user or project identifier, up to 32 bytes long, mapped to uppercase. Must begin with a letter, and contain only letters, digits, or the special characters "\$", ".", or "_".
password	PRIMOS user login password, up to 16 bytes long, mapped to uppercase. May contain any characters except PRIMOS reserved characters.
dec	Decimal integer with optional sign, in the range $(2^{*}31 - 1)$ to $(-2^{*}31 + 1)$ .
oct	Octal integer with optional sign, in the range $(2^{*}31 - 1)$ to $(-2^{*}31 + 1)$ .
hex	Hexadecimal integer, unsigned, in the range 0 to $(2^{*}32 - 1)$ .
date	A calendar date and time in one of the standard formats:  <div style="margin-left: 40px;">ISO        (YY-MM-DD.HH:MM:SS.dow)</div> <div style="margin-left: 40px;">USA        (MM/DD/YY.HH:MM:SS.dow)</div> <div style="margin-left: 40px;">Visual    (DD Mmm YY HH:MM:SS day-of-week)</div> <p>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.</p>
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. Up to 160 bytes long. 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:

<u>Data Type</u>	<u>PL/I Type</u>	<u>FORTTRAN Type</u>
char	char(80) var	INTEGER(41)
charl	char(80) var	INTEGER(41)
tree	char(128) var	INTEGER(65)
entry	char(32) var	INTEGER(17)
id	char(32) var	INTEGER(17)
password	char(16) var	INTEGER(9)
dec	fixed bin(31)	INTEGER*4
oct	fixed bin(31)	INTEGER*4
hex	fixed bin(31)	INTEGER*4
date	fixed bin(31)	INTEGER*4
ptr	ptr options(short)	INTEGER*4
rest	char(160) var	INTEGER(81)
UNCL	char(160) var	INTEGER(81)

Examples are:

<u>Picture</u>	<u>Structure</u>
char; end	dcl 1 struc, 2 char_arg char(80) var;
tree; dec; charl; end	dcl 1 struc, 2 tree_arg char(128) var, 2 dec_arg fixed(31), 2 charl_arg char(80) var;

Use of Data Types REST and UNCL: These two data types cause special processing to occur.

The UNCL data type can 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(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
```

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(31),
    2 date_arg fixed(31);
```

(null command line)

would assign @.LIST (note uppercase conversion) to struc.tree\_arg, 99 to struc.dec\_arg; and 81-01-01.00:00:00 (in file system format) to struc.date\_arg.

Repeat Counts: To save typing, a repeat count feature is included in the syntax. To use it, simply prefix the argument group to be duplicated with the repeat count followed by "\*". For example:

```
5 * dec = -1; 2 * char = foo; end
```

```
dcl 1 struc,
    2 dec_args(5) fixed(31),
    2 char_args(2) char(80) var;
```

The repeat count must be positive and less than 1000.

Note the use of arrays in the structure above. This is not required; one could employ five scalar fixed(31) members with different names in place of dec\_args, for example.

Option Arguments: CL\$PIX allows convenient handling of PRIMOS command line option arguments. An argument group that specifies an option argument is distinguished from an object argument group by beginning with a "-". The general form is:

```
-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(31),
    2 range_upper fixed(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(31);
```

When the option `-STRING` is seen on the command line, the entire remainder of the command is assigned to the `REST` argument, in this case `struc.string_rest`. For example:

```
foo -page 17 -string abc def -page 0
```

assigns `'FOO'` to `struc.char_arg`, `'1'b` to `struc.string_flag`, `'abc def -page 0'` to `struc.string_rest`, `'1'b` to `struc.page_flag`, and `17` to `struc.page_number`.

Note that `CL$PIX` (at least) is not confused by the second occurrence of `-page`: it is part of `struc.string_rest` because it follows the `-string` option.

Using the UNCL Data Type with Option Arguments: The data type `UNCL` may only be assigned to an object argument, not to the parameter of an option argument. However, it is possible for option arguments to be unclaimed and hence added to the `UNCL` argument.

Consider the problem: write a command interface that accepts a `treename` object argument and the option argument `-time` with an integer parameter, but which accepts and passes on all other arguments to some other interface.

A picture to do this is:

```
tree; UNCL; -time dec; end
```

```
dcl 1 struc,
    2 tree_arg char(128) var,
    2 UNCL_arg char(160) var,
    2 time_flag bit(1) aligned,
    2 time_number fixed(31);
```

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 char1; -select char1; -select char1; 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 exactly the same as that accepted by the CPL &ARGS directive. (In fact, CPL uses CL\$PIX in CPL mode to process the &ARGS directive.) 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$get (arg_line, 150, code);
if code ^= 0
  then call errpr$(k$nrtn, 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('3'b3, '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 above 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\$.

# INDEXES

## Index of Subroutines by Name

A\$xy series	FORTTRAN 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	2-15
ALC\$RA	Allocate space for EPF function return information.	III	4-16
ALOC\$\$	Allocate memory on the current stack.	III	4-3
ALS\$RA	Allocate space and set value of EPF function.	III	4-21
AP\$FX\$	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	17-42
ASCSRT	Synonym for ASCS\$\$.		See above.
ASNLN\$	Assign AMLC line.	IV	8-21

ASSUR\$	Check process has given amount of timeslice left.	III	2-22
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	3-8
AT\$HOM	Set the attach point to the home directory.	II	3-10
AT\$LDEV	Set the attach point by top-level directory and logical disk number.	II	3-11
AT\$OR	Set the attach point to the login directory.	II	3-13
AT\$REL	Set the attach point relative to the current directory.	II	3-15
ATCH\$\$	Set the attach point to a specified directory.	II	A-2
ATTDEV	Change a device assignment temporarily.	IV	3-6
BIN\$SR	Perform binary search in ordered table.	III	6-21
BNSRCH	Binary search.	IV	17-48
BREAK\$	Inhibit or enable BREAK function.	III	3-50
BUBBLE	Bubble sort.	IV	17-50
C\$xy series	FORTTRAN compiler conversion functions.	I	B-5
C\$A01	Control functions for user terminal.	IV	6-5
C\$M05	Control functions for 9-track tape.	IV	E-5
C\$M10	Control functions for 7-track tape.	IV	E-5
C\$M11	Control functions for 7-track tape (BCD).	IV	E-5
C\$M13	Control functions for 9-track tape (EBCDIC).	IV	E-5
C\$P02	Control functions for paper tape.	IV	6-12
C1IN	Read a character.	III	3-5
C1IN\$	Read a character.	III	3-7
C1NE\$	Read a character, suppressing echo.	III	3-9
CALAC\$	Determine whether an object is accessible for a given action.	II	2-17
CASE\$A	Convert between upper- and lowercase.	IV	14-2
CAT\$DL	Delete an access category.	II	2-19
CE\$BRD	Return caller's maximum command environment breadth.	II	6-3
CE\$DPT	Return caller's maximum command environment depth.	II	6-4
CH\$FX1	Convert string (decimal) to 16-bit integer.	III	6-3
CH\$FX2	Convert string (decimal) to 32-bit integer.	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	2-23
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	3-10
CL\$PIX	Parse command line according to a command line picture.	II	6-5
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
CLO\$SA	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	18-16
CMDET	Calculate matrix determinant (complex).	IV	18-18
CMDL\$A	Parse a command line.	IV	16-2
CMIDN	Set matrix to identity matrix (complex).	IV	18-20
CMINV	Calculate signed cofactor (complex).	IV	18-22
CMLV\$E	Call new command level after an error.	III	5-5
CMLLT	Matrix multiplication (complex).	IV	18-24
CMSCL	Multiply matrix by scalar (complex).	IV	18-26
CMSUB	Matrix subtraction (complex).	IV	18-28
CMTRN	Calculate transpose matrix (complex).	IV	18-30
CNAM\$S	Change the name of an object in the current directory.	II	4-11
CNIN\$	Read a specified number of characters.	III	3-13
CNSIG\$	Continue scan for on-units.	III	7-19
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	3-52
COM\$AB	Expand a line using Abbreviations preprocessor.	III	2-25
COMANL	Read a line into a PRIMOS buffer.	III	3-15
COMB	Generate matrix combinations.	IV	18-5
COMI\$S	Switch input between the terminal and a file.	III	3-53
COMLV\$	Call a new command level.	III	5-6
COMO\$S	Switch output between the terminal and a file.	III	3-55
CONTRL	Perform device-independent control functions.	IV	4-11
CP\$	Invoke a command from a running program.	II	6-9
CPUID\$	Return model number of Prime computer.	III	2-5

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	10-2
CSUB\$A	Compare two substrings for equality.	IV	10-4
CTIM\$A	Return CPU time since login.	IV	12-2
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	FORTTRAN compiler division functions.	I	B-7
D\$INIT	Initialize disk.	IV	5-13
DATE\$	Return current date and time.	III	2-8
DATE\$A	Return today's 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.	III	10-3
DKGEO\$	Register disk format with driver.	IV	5-18
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	18-16
DMDET	Calculate determinant (double precision).	IV	18-18
DMIDN	Set matrix to identity matrix (double precision).	IV	18-20
DMINV	Calculate inverted matrix (double precision).	IV	18-22
DMMLT	Matrix multiplication (double precision).	IV	18-24
DMSCL	Multiply matrix by a scalar (double precision).	IV	18-26
DMSUB	Matrix subtraction (double precision).	IV	18-28
DMTRN	Calculate transpose matrix (double precision).	IV	18-30
DOFY\$A	Return today's date as day of year (Julian).	IV	12-4
DS\$AVL	Return data about a disk partition.	III	2-51

DS\$ENV	Return data about a process's environment.	III	2-53
DS\$UNI	Return data about file units.	III	2-57
DTIM\$A	Return disk time since login.	IV	12-5
DUPLX\$	Control the way PRIMOS treats the user terminal.	III	3-57
DY\$SGS	Return maximum number of dynamic segments.	III	4-25
E\$xy series	FORTRAN compiler exponentiation routines.	I	B-8
EDAT\$A	Today's date, European (military) style.	IV	12-6
ENCD\$A	Make a number printable if possible.	IV	14-8
ENCRYPT\$	Encrypt login validation passwords.	III	6-24
ENT\$RD	Return the contents of a named entry in a directory open on a file unit.	II	4-37
EPF\$ALLC	Perform the linkage allocation phase for an EPF.	II	5-3
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\$INIT	Perform the linkage initialization phase for an EPF.	II	5-9
EPF\$INVK	Initiate the execution of a program EPF.	II	5-11
EPF\$MAP	Map the procedure images of an EPF file into virtual memory.	II	5-15
EPF\$RUN	Combine functions of EPF\$ALLC, EPF\$MAP, EPF\$INIT, and EPF\$INVK.	II	5-19
EQUAL\$	Generate a filename based on another name.	II	4-39
ERKL\$\$	Read or set the erase and kill characters.	III	3-60
ERRPR\$	Print a standard error message.	III	3-30
ERRSET	Set ERRVEC (a system error vector).	III	10-4
ERTXT\$	Return text associated with error code.	III	2-9
EX\$CLR	Disable signalling of EXIT\$ condition.	III	7-35
EX\$RD	Return state of EXIT\$ signalling.	III	7-36
EX\$SET	Enable signalling of EXIT\$ condition.	III	7-37
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	14-12
FIL\$DL	Delete a file identified by a pathname.	II	4-43
FILL\$A	Fill a string with a character.	IV	10-6



# SUBROUTINES, VOLUME II

FINFO\$	Return information about a specified file unit.	II	4-45
FNCHK\$	Verify a supplied string as a valid filename.	II	4-47
FORCEW	Force PRIMOS to write modified records to disk.	II	4-49
FRES\$RA	De-allocate space for EPF function return information.	III	4-23
FSUB\$A	Fill a substring with a given character.	IV	10-7
FTIM\$A	Convert the TIMMOD field returned by REDN\$\$.	IV	14-14
G\$METR	Return system metering information.	III	2-63
GCHAR	Get a character from an array.	III	6-25
GCHR\$A	Get a character from a packed string.	IV	10-9
GEND\$A	Position to end of file.	IV	15-5
GETERR	Return ERRVEC contents.	III	10-6
GETID\$	Obtain the user-id and the groups to which it belongs.	II	2-21
GINFO	Return PRIMOS II information.	III	2-10
GPAS\$\$	Obtain the passwords of a subdirectory of the current directory.	II	2-23
GPATH\$	Return the pathname of a specified unit, attach point, or segment.	II	4-51
GSNAM\$	Return current PRIMOS system name.	III	2-12
GT\$PAR	Parse character string into tokens.	III	6-27
GV\$GET	Retrieve the value of a global variable.	II	6-12
GV\$SET	Set the value of a global variable.	II	6-14
H\$xy series	FORTTRAN compiler complex number storage.	I	B-5
HEAP	Heap sort.	IV	17-51
I\$AA01	Read ASCII from terminal.	IV	6-8
I\$AA12	Read ASCII from terminal or input stream by REDN\$\$.	IV	6-10
I\$AC03	Input from parallel card reader.	IV	7-22
I\$AC09	Input from serial card reader.	IV	7-24
I\$AC15	Read and print card from parallel card reader.	IV	7-26
I\$AD07	Read ASCII from disk.	IV	5-4
I\$AM05	Read ASCII from 9-track tape.	IV	E-7
I\$AM10	Read ASCII from 7-track tape.	IV	E-7
I\$AM11	Read BCD from 7-track tape.	IV	E-7
I\$AM13	Read EBCDIC from 9-track tape.	IV	E-7
I\$AP02	Read paper tape (ASCII).	IV	6-13
I\$BD07	Read binary from disk.	IV	5-8
I\$BM05	Read binary from 9-track.	IV	E-7
I\$BM10	Read binary from 7-track.	IV	E-7

ICES\$	Initialize the command environment.	III	5-8
IDCHK\$	Validate a name.	III	2-27
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	18-16
IMDET	Calculate matrix determinant (integer).	IV	18-18
IMIDN	Set matrix to identity matrix (integer).	IV	18-20
IMMLT	Matrix multiplication (integer).	IV	18-24
IMSCL	Multiply matrix by scalar (integer).	IV	18-26
IMSUB	Matrix subtraction (integer).	IV	18-28
IMTRN	Calculate transpose matrix (integer).	IV	18-30
IN\$LO	Determine if a forced logout is in progress.	III	2-28
INSERT	Insertion sort.	IV	17-52
IOA\$	Provide free-format output.	III	3-32
IOA\$ER	Provide free-format output, for error messages.	III	3-38
IOA\$RS	Perform free-format output to a buffer.	III	6-32
IOCS\$F	Free logical unit.	IV	3-4
IOCS\$G	Get logical unit.	IV	3-2
ISACL\$	Determine whether an object is ACL-protected.	II	2-25
ISREM\$	Determine whether an open file system object is local or remote.	II	4-54
JSTR\$A	Left-justify, right-justify, or center a string.	IV	10-10
KLM\$IF	Enable a program to obtain serialization data from a specified file.	III	5-8a
L\$xy series	FORTTRAN compiler complex number loading.	I	B-5
LDISK\$	Return information on the system's list of logical disks.	II	4-56
LIMIT\$	Set and read various timers.	III	8-36
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	6-16
LOGO\$\$	Log out a user.	III	2-29
LON\$CN	Switch logout notification on or off.	III	5-20
LON\$R	Read logout notification information.	III	5-21
LOV\$SW	Indicate if the Login-over-login function is currently permitted.	III	2-13
LSTR\$A	Locate one string within another.	IV	10-12
LSUB\$A	Locate one substring within another.	IV	10-14

LUDEV\$	Return a list of devices that a user can access.	III	2-31
LUDSK\$	List the disks a given user is using.	II	4-59
LV\$GET	Retrieve the value of a CPL local variable.	II	6-18
LV\$SET	Set the value of a CPL local variable.	II	6-20
M\$xy series	FORTTRAN 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	10-16
MCOF	Calculate signed cofactor (single precision).	IV	18-13
MCON	Set matrix to constant matrix (single precision).	IV	18-16
MDET	Calculate matrix determinant (single precision).	IV	18-18
MGSET\$	Set the receiving state for messages.	III	9-5
MIDN	Set matrix to identity matrix (single precision).	IV	18-20
MINV	Calculate inverted matrix (single precision).	IV	18-22
MKLB\$F	Convert FORTRAN statement label to PL/I format.	III	7-20
MKON\$F	Create an on-unit (for FTN users).	III	7-21
MKON\$P	Create an on-unit (for any language except FTN).	III	7-23
MKONU\$	Create an on-unit (for PMA and PL/I users).	III	7-25
MM\$MLPA	Make the last page of a segment available.	III	4-4a
MM\$MLPU	Make the last page of a segment unavailable.	III	4-4b
MLLT	Matrix multiplication (single precision).	IV	18-24
MOVEW\$	Move a block of memory.	III	6-34
MRG1\$\$	Merge sorted files.	IV	17-33
MRG2\$	Return next merged record.	IV	17-37
MRG3\$\$	Close merged input files.	IV	17-38
MSCL	Matrix addition (single precision).	IV	18-26
MSG\$ST	Return the receiving state of a user.	III	9-3
MSTR\$A	Move one string to another.	IV	10-18
MSUB	Matrix subtraction (single precision).	IV	18-28
MSUB\$A	Move one substring to another.	IV	10-20
MTRN	Calculate transpose matrix (single precision).	IV	18-30

N\$xy series	FORTTRAN compiler negation functions.	I	B-5
NAMEQ\$	Compare two character strings.	III	6-35
NLEN\$A	Determine the operational length of a string.	IV	10-22
O\$AA01	Write ASCII to terminal or command stream.	IV	6-6
O\$AC03	Parallel interface to card punch.	IV	7-31
O\$AC15	Parallel interface punch and print.	IV	7-32
O\$AD07	Write compressed ASCII to disk.	IV	E-2
O\$AD08	Write ASCII uncompressed to disk.	IV	5-10
O\$ALxx	Interface to various printer controllers.	IV	7-1
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	7-13
O\$AM05	Write ASCII to 9-track tape.	IV	E-7
O\$AM10	Write ASCII to 7-track tape.	IV	E-7
O\$AM11	Write BCD to 7-track tape.	IV	E-7
O\$AM13	Write EBCDIC to 9-track tape.	IV	E-7
O\$BD07	Write binary to disk.	IV	5-6
O\$BM05	Write binary to 9-track tape.	IV	E-7
O\$BM10	Write binary to 7-track tape.	IV	E-7
O\$BP02	Punch paper tape (binary).	IV	6-15
OPEN\$A	Open supplied 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	7-4
OPSR\$S	Locate a file using a search list and a list of suffixes.	II	7-10
OPVP\$A	Read filename and open, or verify and delay.	IV	15-13
OVERFL	Check if an overflow condition has occurred.	III	10-7
P1IB	Input character from paper tape reader to Register A.	IV	6-17
P1IN	Input character from paper tape to variable.	IV	6-19
P1OB	Output character from Register A to paper-tape punch.	IV	6-18
P1OU	Output character from variable to paper-tape punch.	IV	6-20
PA\$DEL	Remove an object's priority access.	II	2-27
PA\$LST	Obtain the contents of an object's priority ACL.	II	2-28
PA\$SET	Set priority access on an object.	II	2-30

PAR\$RV	Return a logical value indicating ACL and quota support.	II	4-61
PERM	Generate matrix permutations.	IV	18-32
PHANT\$	Start a phantom process.	III	10-8
PHNTM\$	Start up a phantom process.	III	5-23
PL1\$NL	Perform a nonlocal GOTO.	III	7-27
POSN\$A	Position file.	IV	15-17
PRERR	Print an error message.	III	10-9
PRI\$RV	Return operating system revision number.	III	2-15
PRJID\$	Return the user's project identifier.	III	2-34
PRWF\$S	Read, write, position, or truncate a file.	II	4-63
PTIME\$	Return amount of CPU time used since login.	III	2-35
PWCHK\$	Validate syntax of a password.	III	2-36
Q\$READ	Return directory quota and disk record usage information.	II	4-70
Q\$SET	Set a quota on a subdirectory of the current directory.	II	4-73
QUICK	Partition exchange sort.	IV	17-54
QUIT\$	Determine if there are pending quits.	III	3-62
RADSEX	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 depth.	II	6-22
RDASC	Read ASCII from any device.	IV	4-5
RDBIN	Read binary from any device.	IV	4-9
RDEN\$S	Position in or read from a directory.	II	A-9
RDLIN\$	Read a line of characters from a compressed ASCII disk file.	II	4-76
RDTK\$S	Parse a command line.	III	3-16
READY\$	Display PRIMOS command prompt.	III	2-37
REMEPF\$	Remove an EPF from a user's address space.	II	5-22
REST\$S	Restore an R-mode executable image.	III	5-13
RESU\$S	Restore and resume an R-mode executable image.	III	5-15
RLSE\$S	Get input records after SETU\$.	IV	17-26
RMSGD\$	Receive a deferred message.	III	9-7
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	5-24

RPOSSA	Return position of file.	IV	15-18
RRECL	Read disk record.	IV	5-14
RSEGAC\$	Determine access to a segment.	III	2-16
RSTR\$A	Rotate string left or right.	IV	10-23
RSUB\$A	Rotate substring left or right.	IV	10-26
RTRN\$\$	Get sorted records.	IV	17-28
RVON\$F	Revert an on-unit (for FTN users).	III	7-28
RVONU\$	Revert an on-unit (for any language except FTN).	III	7-29
RWND\$A	Reposition file.	IV	15-19
S\$xy series	FORTTRAN compiler subtraction routines.	I	B-8
SATR\$\$	Set or modify an object's attributes.	II	4-78
SAVE\$\$	Save an R-mode executable image.	III	5-17
SCHAR	Store a character into an array location.	III	6-37
SEM\$CL	Release (close) a named semaphore.	III	8-17
SEM\$DR	Drain a semaphore.	III	8-19
SEM\$NF	Notify a semaphore.	III	8-21
SEM\$OP	Open a set of named semaphores.	III	8-23
SEM\$OU	Open a set of named semaphores.	III	8-23
SEM\$TN	Periodically notify a semaphore.	III	8-27
SEM\$TS	Return number of processes waiting on a semaphore.	III	8-29
SEM\$TW	Wait on a specified named semaphore, with timeout.	III	8-31
SEM\$WT	Wait on a semaphore.	III	8-33
SETRC\$	Record command error status.	III	5-9
SETU\$\$	Prepare sort table and buffers for CMBN\$.	IV	17-22
SGD\$DL	Delete a segment directory.	II	4-84
SGD\$EX	Find out if there is a valid entry at the current position within the segment directory on a specified unit.	II	4-86
SGD\$OP	Open a segment directory entry.	II	4-88
SGDR\$\$	Position, read, or modify a segment directory.	II	4-90
SGNL\$F	Signal a condition.	III	7-30
SHELL	Diminishing increment sort.	IV	17-56
SID\$GT	Return user number of initiating process.	III	2-38
SIGNL\$	Signal a condition.	III	7-32
SIZE\$	Return the size of a file system entry.	II	4-96
SLEEP\$	Suspend a process for a specified interval.	III	8-39
SLEP\$I	Suspend a process (interruptible).	III	8-40
SLITE	Set the sense light on or off.	III	10-12
SLITET	Test sense light settings.	III	10-13
SMSG\$	Send an interuser message.	III	9-9
SNCHK\$	Check validity of system name passed to it.	III	2-18
SP\$REQ	Insert a file into the spool queue.	IV	7-12c

SPAS\$\$	Set the owner and nonowner passwords on an object.	II	2-32
SPOOL\$	Insert a file in spooler queue.	IV	7-8
SR\$ABSDS	Disable optional rules enabled by SR\$ENABL.	II	7-17
SR\$ADDB	Add a rule to the start of a search list or before a specified rule within the list.	II	7-20
SR\$ADDE	Add a rule to the end of a search list or after a specified rule within the list.	II	7-23
SR\$CREAT	Create a search list.	II	7-26
SR\$DEL	Delete a search list.	II	7-28
SR\$DSABL	Disable an optional search rule enabled by SR\$ENABL.	II	7-30
SR\$ENABL	Enable an optional search rule.	II	7-33
SR\$EXSTR	Determine if a search rule exists.	II	7-36
SR\$FR_LS	Free list structure space allocated by SR\$LIST or SR\$READ.	II	7-40
SR\$INIT	Initialize all search lists to system defaults.	II	7-42
SR\$LIST	Return the names of all defined search lists.	II	7-44
SR\$NEXTR	Read the next rule from a search list.	II	7-48
SR\$READ	Read all of the rules in a search list.	II	7-53
SR\$REM	Remove a rule from a search list.	II	7-57
SR\$SETL	Set the locator pointer for a search rule.	II	7-60
SR\$SSR	Set a search list via a user-defined search rules file.	II	7-63
SRCH\$\$	Open, close, delete, or verify existence of an object.	II	4-99
SRSFX\$	Search for a file with a list of possible suffixes.	II	4-108
SRTF\$\$	Sort several input files.	IV	17-16
SS\$ERR	Signal an error in a subsystem.	III	5-11
SSTR\$A	Shift string left or right.	IV	10-28
SSUB\$A	Shift substring left or right.	IV	10-30
SSWICH	Test sense switch settings.	III	10-14
ST\$SGS	Return maximum number of static segments.	III	4-26
STR\$AL	Allocate user-class dynamic memory.	III	4-5
STR\$AP	Allocate process-class dynamic memory.	III	4-7
STR\$AS	Allocate subsystem-class dynamic memory.	III	4-8
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	17-40
SUSR\$	Test if current user is supervisor.	III	2-39

T\$AMLC	Communicate with AMLC driver.	IV	8-23
T\$CMPC	Input from MPC card reader.	IV	7-28
T\$LMPC	Move data to LPC line printer.	IV	7-6
T\$MT	Raw data mover for tape.	IV	7-37
T\$PMPC	Raw data mover for card reader.	IV	7-34
T\$SLC0	Communicate with SMLC driver.	IV	8-3
T\$VG	Interface to Versatec printer.	IV	7-16
TLIB	Read a character (function) from PMA into Register A.	III	3-23
TlIN	Read a character (procedure).	III	3-24
TlOB	Write one character from Register A.	III	3-47
TlOU	Write one character.	III	3-48
TEMP\$A	Open a scratch file.	IV	15-20
TEXTOS\$	Check filename for valid format.	III	10-15
TI\$MSG	Display standard message showing times used.	III	2-40
TIDEC	Read a decimal number.	III	3-26
TIHEX	Read a hexadecimal number.	III	3-27
TIMDAT	Return timing information and user identification.	III	2-42
TIME\$A	Return time of day.	IV	12-7
TIOCT	Read an octal number.	III	3-28
TL\$SGS	Return highest segment number.	III	4-27
TNCHK\$	Verify a supplied string as a valid pathname.	II	4-114
TNOU	Write characters to terminal, followed by NEWLINE.	III	3-40
TNOUA	Write characters to terminal.	III	3-41
TODEC	Write a signed decimal number.	III	3-42
TOHEX	Write a hexadecimal number.	III	3-43
TONL	Write a NEWLINE.	III	3-44
TOOCT	Write an octal number.	III	3-45
TOVFD\$	Write a decimal number, without spaces.	III	3-46
TREE\$A	Test for pathname.	IV	10-32
TRNC\$A	Truncate a file.	IV	15-22
TSCN\$A	Scan the file system tree structure.	IV	15-23
TSRC\$S	Open, close, delete, or find a file anywhere in the file structure.	II	A-17
TTY\$IN	Check for unread terminal input characters.	III	3-63
TTY\$RS	Clear the terminal input and output buffers.	III	3-65
TYPE\$A	Determine string type.	IV	10-35
UID\$BT	Return unique bit string.	III	6-39
UID\$CH	Convert UID\$BT output into character string.	III	6-40
UNIT\$A	Check for file open.	IV	15-28
UNIT\$S	Return caller's minimum and maximum file unit numbers.	II	4-117
UNO\$GT	List users with same name as caller.	III	2-44



# SUBROUTINES, VOLUME II

UPDATE	Update current directory (PRIMOS II only.	III	10-17
USERS\$	Return user number and count of users.	III	2-20
UTYPE\$	Return user type of current process.	III	2-45
VALID\$	Validate a name against composite identification.	III	2-48
WILD\$	Return a logical value indicating whether a wildcard name was matched.	II	4-118
WRASC	Write ASCII.	IV	4-3
WRBIN	Write binary to any output device.	IV	4-7
WRECL	Write disk record.	IV	5-17
WTLIN\$	Write a line of characters to a compressed ASCII file.	II	4-120
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, 2-19

- calculating accessibility, 2-17
- category, 2-3, 2-15, 2-16
- changing, 2-5, 4-99, 4-102
- copying, of another object, 2-9
- default, 2-7
- listing, 2-11
- modifying, 2-5
- nonowner, 2-18, 2-23, 2-32
- owner, 2-18, 2-23, 2-32
- priority, 2-27 to 2-30
- specific, 2-15

Access Category, searching for, 7-4, 7-11

Access control, 2-1 to 2-33

- ACL structure, 2-12
- converting ACL to password, 2-13
- group-id, 2-12, 2-21
- user-id, 2-12, 2-21

ACL protection and quotas, supporting, 4-61

### Adding search rule,

- after existing rule, 7-24
- before existing rule, 7-21
- to beginning of search list, 7-21
- to end of search list, 7-24

Addressing modes and libraries, 1-14

Administrator search rules, 7-21, 7-58

### Arguments,

- how to set bits in, 1-11
- keys as, 1-12
- parsing command, 6-5, C-1 to C-13

Attach point, setting (See Attaching)

Attaching, 3-1 to 3-17, A-2

- relative to current directory, 3-15
- to any directory, 3-3
- to home directory, 3-10
- to login directory, 3-13
- to origin directory, 3-13
- to top-level directory, 3-6, 3-8, 3-11

Attribute,  
 date and time, 4-81  
 dumped, 4-81  
 password protection, 4-79  
 read/write lock, 4-81  
 setting directory, 4-15, A-5,  
 A-7  
 setting file, 4-78

## B

BIN data types, 1-7  
 BIT data types, 1-7  
 Bits,  
 how to set in arguments, 1-11  
 positional values of, 1-11,  
 1-12

## C

CALL statement, 1-4, 1-5  
 Change open mode, 4-6  
 CHAR data types, 1-7  
 Closing file system objects, 4-7  
 to 4-10, 4-99, 4-102, 4-105,  
 A-17  
 Command arguments, parsing, 6-5,  
 C-1 to C-13  
 Command environment, 6-1 to 6-22  
 current depth of, 6-22  
 maximum breadth of, 6-3  
 maximum depth of, 6-4  
 Command invocation from a running  
 program, 6-9  
 Command state flags, EPF, 5-5

Creating,  
 directory, 4-15, A-5, A-7  
 file system object, 4-99,  
 4-105, A-17  
 file using search rules, 7-4,  
 7-11  
 search list, 7-27, 7-64

## D

Data types,  
 equivalents table, B-2  
 FIXED BIN, 1-7  
 FLOAT BIN, 1-7  
 parameter, 1-7  
 returned value, 1-7  
 Deactivating an EPF, 5-7  
 Declaration, 1-4, 1-5  
 arrays in, 1-8  
 DECLARE (DCL) statement, 1-4,  
 1-5  
 function, 1-5  
 structures in, 1-9  
 subroutine, 1-4  
 Deleting,  
 file system objects, 4-43,  
 4-99, 4-105, A-17  
 multiple search rules, 7-64  
 search list, 7-29  
 search rule, 7-58  
 segment directory entry, 4-84  
 Directory,  
 attaching to, (See also  
 Attaching)  
 creating, 4-15, A-5, A-7  
 deleting, 4-43  
 listing, 4-17  
 quota, 4-70, 4-73  
 reading entries in, 4-24  
 searching for, 7-4, 7-11  
 searching through, 4-17, 4-24,  
 4-29, 4-37, A-9  
 segment, 4-90  
 selecting entries in, 4-29,  
 4-37  
 Disabling optional search rule,  
 7-18, 7-31

E

Enabling optional search rule,  
7-34

EPF,

- command state flags, 5-5
- deactivating, 5-7
- initializing, 5-9, 5-19
- invoking, 5-11, 5-19
- linkage allocation, 5-3, 5-19
- managing, 5-1 to 5-25
- mapping, 5-15, 5-19
- removing, 5-22
- replacing, 5-24

Error code, standard, 1-13

Existence of an object,  
verifying, 4-99, 4-103, A-17

F

File,

- creating with search rules,  
7-4, 7-11
- deleting, 4-43
- forced writing of, 4-49, 4-65
- locating, 4-54
- opening with search rules,  
7-4, 7-11
- opening with search rules and  
suffix list, 7-11
- positioning, 4-63
- reading, 4-63, 4-76
- searching for, 7-4, 7-11
- truncating, 4-63
- writing, 4-49, 4-63, 4-120

File attributes, setting, 4-78

File name,

- extracting from pathname, 4-41
- generating, 4-39
- verifying string as, 4-47

File system object,

- changing, 4-11
- changing access to, 4-102
- closing, 4-99, 4-102, 4-105
- creating, 4-99, 4-105
- deleting, 4-43, 4-99, 4-105

File system object (continued)

- locating, 4-54
- opening, 4-99, 4-101, 4-105
- size of, 4-96
- verifying existence of, 4-99,  
4-103

File unit,

- assignment, 7-4, 7-11
- closing file by, 4-10
- number, minimum and maximum,  
4-114
- obtaining information about,  
4-45
- obtaining pathname from, 4-51

FIXED BIN data types, 1-7

FLOAT BIN data types, 1-7

Forced writing of files, 4-49,  
4-65

Free allocated space, 7-41

Function,

- call, 1-5
- declaration, 1-5
- defined, 1-1
- distinguished from subroutine,  
1-1
- without parameters, 1-5

G

Generation, file name, 4-39

Global variable,

- retrieving value of, 6-12
- setting value of, 6-14

H

Home\_dir (See Search rule  
keywords)

I

Initializing,

EPF, 5-9, 5-19  
search lists, 7-43

Invoking,

command from a running program,  
6-9  
EPF, 5-11, 5-19

K

Keys as arguments, 1-12

L

Libraries,

and addressing modes, 1-14  
subroutine, 1-14

Linkage allocation, EPF, 5-3,  
5-19

Listing,

commands at mini-command level,  
6-16  
directory entries, 4-17

Local objects, 4-54

Local variable,

retrieving value of, 6-18  
setting value of, 6-20

Locating objects, 4-54

Logical disks,

in use by caller, 4-59  
system's list of, 4-56

M

Mapping an EPF, 5-15, 5-19

Mini-command level, listing  
commands at, 6-16

Modes and libraries, 1-14

O

Object name, changing, 4-11

Open mode, changing, 4-6

Opening,

file system object, 4-99,  
4-101, 4-105, A-17  
file using search rules, 7-4,  
7-11  
file using search rules and  
suffix list, 7-11  
segment directory entry, 4-88

Optional search rule,

checking existence of, 7-37  
disabling, 7-18, 7-31  
enabling, 7-34  
reading disabled, 7-54  
skipping disabled, 7-49  
status of, 7-54

Origin\_dir (See Search rule  
keywords)

P

Parameter, 1-6 to 1-10  
data types, 1-7  
optional, 1-9

Parsing command arguments, 6-5,  
C-1 to C-13

Pathname,

obtaining, from file unit,  
4-51  
verification of string as,  
4-114

POINTER data types, 1-8

Positioning,

file, 4-63  
segment directory, 4-90

Q

Quota, directory, 4-70, 4-73

Quotas and ACL protection,  
supporting, 4-61

R

Reading,  
  directory entries, 4-24  
  files, 4-63, 4-76  
  search rule, 7-54  
  segment directory entries,  
    4-90

Referencing\_dir (See Search rule  
  keywords)

Remote objects (See Locating  
  objects)

Removing,  
  EPF, 5-22  
  search rule, 7-58

Replacing an EPF, 5-24

Returned value,  
  data types, 1-7  
  optional, 1-10  
  RETURNS descriptor, 1-5

S

Satisfying references,  
  at load time, 1-14  
  at run time, 1-15

Search list,  
  appending to, 7-64  
  'blank', 7-27  
  checking for search rule, 7-37  
  creating, 7-27  
  deleting, 7-29  
  deleting multiple, 7-43  
  displaying names of, 7-45  
  initializing all, 7-43  
  reading all rules in, 7-54  
  reading next rule in, 7-49

Search list (continued)  
  search rules file name, 7-45  
  setting, 7-64  
  template file name, 7-45

Search rule keywords,  
  checking existence of, 7-37  
  current values of, 7-49  
  home\_dir, 7-24, 7-37, 7-38,  
    7-50  
  -insert, 7-65  
  origin\_dir, 7-22, 7-24, 7-25,  
    7-37, 7-38, 7-50  
  referencing\_dir, 7-4, 7-11,  
    7-24, 7-37, 7-38, 7-49 to  
    7-51  
  -system, 7-65

Search rule locator,  
  reading, 7-49  
  setting, 7-61

Search rule subroutines, 7-1  
  (See also Search rule keywords;  
    Search rule locator; Search  
    rules; Search rules file)  
  case sensitivity, 7-3, 7-37  
  data types, 7-3  
  list of, 7-2  
  wildcards, 7-3

Search rules, (See also Search  
  rules file)  
  adding multiple, 7-64  
  adding to search list, 7-21,  
    7-24  
  checking existence of, 7-37  
  deleting from list, 7-58  
  deleting multiple, 7-64  
  determining type, 7-49  
  disabling optional, 7-18, 7-31  
  enabling optional, 7-34  
  reading disabled, 7-54  
  reading from list, 7-49  
  reading multiple, 7-54  
  system defaults, 7-43

Search rules file,  
  displaying names of, 7-45  
  using, 7-64

SEARCH\_RULES\*, 7-43

Searching,  
 for directory, 7-4, 7-11  
 for file system object, 4-99,  
 A-17  
 for suffixed pathname, 4-108  
 through directory, 4-17, 4-24,  
 4-29, 4-37, A-9

Segment directory,  
 creating with search rules,  
 7-4, 7-11  
 deleting entry, 4-84  
 determining validity of entry,  
 4-86  
 expanding, 4-90  
 opening entry, 4-88  
 positioning in, 4-90  
 reading entry in, 4-90  
 searching for, 7-4, 7-11  
 truncating, 4-90

Selecting directory entries,  
 4-29, 4-37

Setting,  
 bits in arguments, 1-11  
 search list, 7-64

Setting attach point (See  
 Attaching)

Standard error code, 1-13

Subroutine,  
 call, 1-4  
 contents of, 1-2  
 declaration, 1-4  
 defined, 1-1  
 distinguished from function,  
 1-1  
 example of, figure, 1-3  
 libraries, 1-14  
 loading and linking  
 information, 1-14  
 overview, 1-1 to 1-15  
 parameter section, 1-6  
 parameters, 1-6  
 usage section, 1-2

Suffix,  
 appending to pathname, 4-4  
 searching for file with, 4-108  
 using list for searching, 7-11

## T

Template file, 7-45, 7-64

Truncating,  
 file, 4-63  
 segment directory, 4-90

## U

Usage section,  
 data types in, 1-6  
 explained, 1-2

## V

Verifying,  
 existence of object, 4-99,  
 4-103, 4-108, A-17  
 string as filename, 4-47  
 string as pathname, 4-114

## W

Wildcard, matching name with,  
 4-118

Writing files, 4-49, 4-63, 4-120

# **SURVEY**



READER RESPONSE FORM

DOC10081-1LA

Subroutines Reference Guide Volume II

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

\_\_\_excellent \_\_\_very good \_\_\_good \_\_\_fair \_\_\_poor

2. Please rate the document in the following areas:

Readability: \_\_\_hard to understand \_\_\_average \_\_\_very clear

Technical level: \_\_\_too simple \_\_\_about right \_\_\_too technical

Technical accuracy: \_\_\_poor \_\_\_average \_\_\_very good

Examples: \_\_\_too many \_\_\_about right \_\_\_too few

Illustrations: \_\_\_too many \_\_\_about right \_\_\_too few

3. What features did you find most useful? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

4. What faults or errors gave you problems? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Name: \_\_\_\_\_ Position: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_ Zip: \_\_\_\_\_



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

**BUSINESS REPLY MAIL**

Postage will be paid by:



Attention: Technical Publications  
Bldg 10  
Prime Park, Natick, Ma. 01760



READER RESPONSE FORM

DOC10081-1LA

Subroutines Reference Guide Volume II

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

\_\_\_excellent \_\_\_very good \_\_\_good \_\_\_fair \_\_\_poor

2. Please rate the document in the following areas:

Readability: \_\_\_hard to understand \_\_\_average \_\_\_very clear

Technical level: \_\_\_too simple \_\_\_about right \_\_\_too technical

Technical accuracy: \_\_\_poor \_\_\_average \_\_\_very good

Examples: \_\_\_too many \_\_\_about right \_\_\_too few

Illustrations: \_\_\_too many \_\_\_about right \_\_\_too few

3. What features did you find most useful? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

4. What faults or errors gave you problems? \_\_\_\_\_

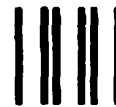
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Name: \_\_\_\_\_ Position: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_ Zip: \_\_\_\_\_



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

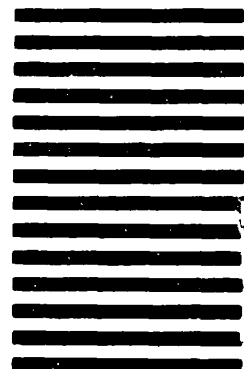
First Class Permit #531 Natick, Massachusetts 01760

**BUSINESS REPLY MAIL**

Postage will be paid by:



Attention: Technical Publications  
Bldg 10  
Prime Park, Natick, Ma. 01760



READER RESPONSE FORM

DOC10081-1LA

Subroutines Reference Guide Volume II

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

\_\_\_excellent \_\_\_very good \_\_\_good \_\_\_fair \_\_\_poor

2. Please rate the document in the following areas:

Readability: \_\_\_hard to understand \_\_\_average \_\_\_very clear

Technical level: \_\_\_too simple \_\_\_about right \_\_\_too technical

Technical accuracy: \_\_\_poor \_\_\_average \_\_\_very good

Examples: \_\_\_too many \_\_\_about right \_\_\_too few

Illustrations: \_\_\_too many \_\_\_about right \_\_\_too few

3. What features did you find most useful? \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

4. What faults or errors gave you problems? \_\_\_\_\_

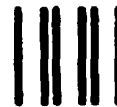
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Name: \_\_\_\_\_ Position: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_ Zip: \_\_\_\_\_



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

**BUSINESS REPLY MAIL**

Postage will be paid by:



**Prime**<sup>TM</sup>

Attention: Technical Publications  
Bldg 10  
Prime Park, Natick, Ma. 01760

