Subroutines Reference Guide Volume III

DOC10082-1LA

Subroutines Reference Guide Volume III

First Edition

by

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 20.2 (Rev. 20.2).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

COPYRIGHT INFORMATION

CREDITS

WITH SPECIAL THANKS TO DAVID BROOKS FOR HIS TECHNICAL EXPERTISE AND TO CAMILLA HAASE FOR WRITING CHAPTERS ONE AND SIX

Project Support	Joan Karp Margaret Taft Alice Landy Richard Frost Leonard Bruns
Editorial Support	Mary Skousgaard Thelma Henner
Graphic Support	Marjorie Clark Mingling Chang
Production Support	Judy Gordon
Document Preparation Support	Celeste Henry Kathy Normington

PRINTING HISTORY -- Subroutines Reference Guide, Volumes I-IV

Edition	Date	Number	Software Release
First Edition	March 1979	PDR3621	16.3
Second Edition	January 1980	PDR3621	17.2
Update 1	December 1980	PTU2600-078	18.1
Third Edition	July 1982	DOC3621-190	19.0
First Edition			
Volume I	August 1986	DOC10080-1LA	20.2
Volume II	August 1986	DOC10081-1LA	20.2
Volume III	August 1986	DOC10082-1LA	20.2
Volume IV	August 1986	DOC10083-1LA	20.2

CUSTOMER SUPPORT CENTER

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

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

HOW TO ORDER TECHNICAL DOCUMENTS

Follow the instructions below to obtain a catalog, price list, and information on placing orders.

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.

Contents

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 executable 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 follows.

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, CBL, 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, and the use of a number of command environment functions. Three chapters are devoted to subroutines related to the file system, and one chapter each is devoted to those 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 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

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.

SUGGESTED REFERENCES

The Prime User's Guide (DOC4130-4LA) contains information on system use, directory structure, the condition mechanism, CPL files, ACLs, and how to load and execute files with external subroutines. Language programmers will also need the reference guide for their particular languages.

Programmers who wish more advanced information on library management or I/O manipulation should consult the System Administrator's Guide (DOC5037-4LA).

The following related Prime publications are also available:

Advanced Programmers's Guide, Volume 0 (DOC9229-1LA)

Instruction Sets Guide (DOC9474-1LA)

Operator's Guide to System Commands (DOC9304-2LA and UPD9304-21A)

SEG and LOAD Reference Guide (DOC3524-192L)

System Architecture Reference Guide (DOC3060-192L)

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id
Abbreviations	If a command or statement has an abbreviation, it is indicated by underlining. In cases where the command or directive itself contains an underscore, the abbreviation is shown below the full name, and the name and abbreviation are placed within braces.	LOGOUT { SET_QUOTA SQ }

Convention	Explanation	Example
Underlining in examples	In examples, user input is underlined but system prompts and output are not.	OK, RESUME MY_PROG This is the output of MY_PROG.CPL OK,
Brackets	Brackets enclose a list of two or more optional items. Choose none, one, or more of these items.	SPOOL [-LIST -CANCEL]
Braces	Braces enclose a list of items. Choose one and only one of these items.	CLOSE { filename ALL }
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	DIM array (row,col)
Hyphen -	Wherever a hyphen appears as the first letter of an option, it is a required part of that option.	SPOOL -LIST

FILENAME CONVENTIONS

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

See the manual for each language for an explanation of how the various names for source, object, listing, and runtime files relate to each other. Also see the Prime User's Guide for a general explanation.

Note

On some devices, the underscore (_) may print as a back arrow (<-).

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

1
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 DELE$A is assigned to the variable VALUE1:

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

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

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

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

SUBROUTINE DESCRIPTIONS

In this guide, each description of a subroutine contains the following sections:

- 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.

Figure 1-1 shows an example of a subroutine description.

SUBROUTINE USAGE

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

1. How to declare the subroutine in a program
2. 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.

UID$BT

Purpose

Returns a unique bit string for identification purposes.

Usage

```
DCL UID$BT ENTRY (BIT (48) ALIGNED);

CALL UID$BT (unique_bit_string);
```

Parameters

unique_bit_string

OUTPUT. Unique bit string returned.

Discussion

The string is guaranteed to be unique. This bit string is not random; it is formed by concatenating the current date and time (in FS format) with a 16-bit counter. (The format of a 32-bit encoded FS-format date is described in Appendix C.) If a random number is required rather than a unique identifier, the applications library routine RAND$A should be 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.

A Subroutine Description
Figure 1-1

Subroutine Declarations

The following example shows a subroutine declaration:

```
DCL CNIN$ ENTRY(CHARACTER(*), FIXED BIN, FIXED BIN);
```

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

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

Subroutine Calls

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

```
CALL CNIN$(buffer, char_count, actual_count);
```

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

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

```
CALL TONL;
```

Function Declarations

The following example shows a function declaration:

```
DCL PWCHK$ ENTRY(FIXED BIN, CHAR(*) VAR) 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:

```
password_ok = PWCHK$(key, password);
```

The equal sign = is the assignment operator. password_ok is a logical (Boolean) variable that is assigned the value returned by the call to PWCHK$. key and password represent integer and character-string values, respectively.

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 called from FTN programs require parameters.

SUBROUTINE PARAMETERS

Subroutines usually expect one or more arguments from the calling program. These arguments must be of the data type specified in the parameter list of the DECLARE statement, and must be passed in the order expected. All standard Prime subroutines are written in FORTRAN, PMA, or a system version of PL/I. Volume I discusses how to translate the data types expected by these languages 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. If too few arguments are passed, execution causes an error message such as POINTER FAULT or ILLEGAL SEGNO. If too many arguments are passed, the subroutine ignores the extra arguments, but will probably perform incorrectly. 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

    1. The parameter, an input parameter, is a pointer.
    2. 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(n)	Also specified as CHARACTER(n), CHARACTER(n) NONVARYING. Specifies a character string or array of length n. A CHAR(n) string is stored as a byte-aligned string, one character per byte. (A byte is 8 bits.)
CHAR(*)	Also CHARACTER(*), CHARACTER(*) NONVARYING. Specifies a character string or array whose length is unknown at the time of declaration. A CHAR(*) string is stored as a byte-aligned string, one character per byte.
CHAR(n) VAR	Also CHARACTER(n) VARYING. Specifies a character string or array whose length can be a maximum of n characters. The first 2 bytes (one halfword) of storage for a CHAR(n) VAR string contain an integer that specifies the string length; these are followed by the string, one character per byte.
CHAR(*) VAR	Also CHARACTER(*) VARYING. Specifies a character string or array whose length is unknown at the time of declaration. The first 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.
(n) FIXED BIN	An integer array of n elements. See below for more information about arrays.
FLOAT BIN	Also FLOAT BIN(23), FLOAT. Specifies a 32-bit (one-word) floating-point number.
FLOAT BIN(47)	Specifies a 64-bit (double-word) floating-point number.

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

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

POINTER Also PTR. Specifies a POINTER data type. A pointer is usually stored in three halfwords (48 bits). If the pointer only points 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 CH$FX1 ENTRY(CHAR(*) VAR, FIXED BIN(15)
                                [, FIXED BIN(15)]);

CALL CH$FX1(string_to_convert, result
                                [, nonstandard_code]);
```

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

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

If echo_flag is missing, the subroutine acts as if it had been supplied with a value of "true." If term_flag is missing, the subroutine acts as if it had been supplied with a value of "false."

In still other cases, the subroutine changes its behavior depending on the presence of the parameter. For example, the subroutine CH$FX1 (described in this volume) 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 is 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$GET, described in this volume. It reads a line from the command device and, in addition, returns a flag that indicates whether a command input file is active. Most programs do not need to know whether a command input file is active. They call CL$GET as a subroutine:

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

A program interested in command input files, however, calls 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.

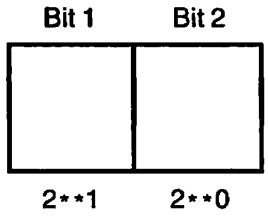
The Usage section of the subroutine descriptions gives both the function invocation and the subroutine invocation for subroutines that are likely to be called in both ways.

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.

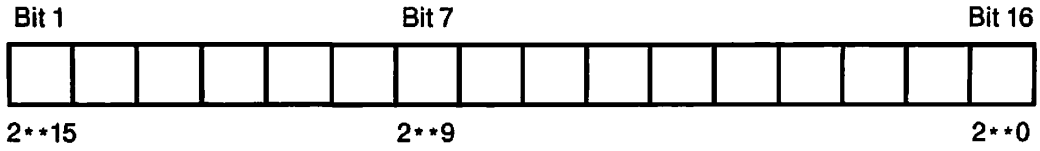
A data item is stored in a computer as a collection of bits, which can each have one of two values, off or on. On Prime computers, off is arbitrarily equated to the bit value '0'B or false, and on is equated to '1'B or true. (This is not the same as the FORTRAN values .FALSE. and .TRUE., which are the LOGICAL data types 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 the bit is on, is 1. The high-order bit is in the position of 2 to the first power, and its value, if the bit is on, is 2. (If the bit is off, its value 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.



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. For explanations of the standard error codes, see Volume 0 of the Advanced Programmer's Guide. Volume I contains a listing of the standard error codes and the messages to which they correspond.

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, are grouped together. There is a separate library for each of these subjects.

Prime computers offer several addressing modes to provide software compatibility to the user. (For a discussion of addressing modes, see the System Architecture Reference Guide.) To maintain this compatibility, a given subroutine library normally exists in three general versions: R-mode, V-mode, and V-mode (unshared). (See Chapter 2 of Volume I of the Subroutines Reference Guide for a discussion of shared and unshared libraries.)

A program is compiled in one of the segmented modes (V-mode or I-mode) or in the older R-mode. If the program is compiled in one of the segmented modes, it may call library routines written in any of the segmented modes. A single set of libraries is provided for all three modes. If the program is compiled in either V-mode or I-mode, it requires the suitable version of the library (normally a V-mode library services both V-mode and I-mode programs). If the program is compiled in R-mode, the program must use the R-mode version of that library.

Every routine description contains a section entitled Loading and Linking Information, which describes how to access the routine from the different modes.

Satisfying the References at Load Time

When the subroutines in this volume 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 routine 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 all the routines described in this volume, only the V-mode or I-mode subroutines with unshared code require a special library. Both the shared code and R-mode code require "no special action." This means that the LI[brary] command with no arguments, which normally ends a loading sequence, satisfies the references.

Getting the Subroutines at Runtime

When a subroutine is available to be shared among 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 retains 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 executable program format (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 herein are providing PRIMOS services, there is no way of providing them as unshared code, since PRIMOS by definition is shared.

For a further description of libraries and related terminology, see Volume I of the Subroutines Reference Guide.

2
Core Operating System Services

This chapter contains routines that provide core operating system services to the programmer.

The first part of this chapter presents routines involving general operating system information. The second part of this chapter describes routines involving system information specific to the current user.

SYSTEM INFORMATION ROUTINES

This section describes the following subroutines:

Routine	Function
ABSW	Returns cold-start setting of ABBREV switch.
CKDYN$	Determines if routine is dynamically accessible.
CPUID$	Returns model number of Prime computer.
DATE$	Returns current date and time.
ERTXT$	Returns text representation of error code.
GINFO	Returns PRIMOS II information.
PRI$RV	Returns operating system revision number.
RSEGAC$	Determines access to a segment.
USER$	Returns user number and count of users.

ABSW

Purpose

This procedure returns the cold-start setting of the abbreviations enable switch.

Usage

```
DCL AB$SW$ ENTRY RETURNS (FIXED BIN);

ab_sw = AB$SW$ ();
```

Parameters

ab_sw

RETURNED VALUE. Returns 1 if the command line abbreviation expansion feature is globally enabled. Returns 0 if the feature is globally disabled. If the feature is enabled, individual users may still elect to disable it.

Discussion

This function cannot be called from FTN because it has no arguments.

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.

CKDYN$

Purpose

This routine accepts a dynamic entrypoint (DYNT) name and determines whether that routine is currently accessible through the PRIMOS dynamic linking mechanism.

Usage

```
DCL ROUTINE_NAME ENTRY (CHAR(32) VAR, FIXED BIN);

CALL CKDYN$ (routine_name, code);
```

Parameters

routine_name1

INPUT. The name of the dynamic entrypoint.

code

OUTPUT. Standard error code. Possible values are:

0	Dynamic entrypoint routine_name was found.
E$FNTF	Dynamic entrypoint routine_name was not found.

Discussion

CKDYN$ looks for the entrypoint in PRIMOS, and in all executable program format (EPF) libraries and static-mode shared libraries currently listed in the user's ENTRY$ search list. If a library does not appear in the ENTRY$ search list, its entrypoints are not accessible to CKDYN$.

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.

CPUID$

Purpose

This routine determines which Prime computer model the program is running on.

Usage

```
DCL CPUID$ ENTRY (POINTER, FIXED BIN);

CALL CPUID$ (struc_ptr, code);
```

Parameters

struc_ptr

INPUT -> OUTPUT. This parameter points to a structure of user memory with the following layout:

```
1  structure,
   2  version fixed bin,
   2  cpu_model fixed bin(31),
   2  microcode,
      3  res1 bit(8),
      3  mfg_rev bit(8),
      3  eng_rev fixed bin,
   2  proc_options,
      3  res2 bit(15),
      3  info_series bit,
   2  res3 bin(31),
   2  res4 bin(31);
```

The fields are defined as follows:

version	Input value. Specifies which version of the structure the caller is expecting to receive. Must be 1.
cpu_model	Code value indicating the processor model number. See Discussion below for a list of the possible values.
res1	Reserved.
mfg_rev	Manufacturing revision number of microcode installed.

eng_rev	Engineering revision number of microcode installed.
res2	Reserved.
info_series	If 1, indicates the processor has special microcode assist for Prime INFORMATION. If 0, indicates the processor has no such microcode assist.
res3, res4	Reserved.

code

OUTPUT. Standard error code. Possible values are:

0	No error.
E$BPAR	version is not 1.

Discussion

At Rev 20.2, the following values can be returned in version:

Value	Processor model
0	P400 with rev A microcode, or original P500
1	P400 with rev B or later microcode
3	P350
4	P250-II, P450, or P550-I
5	P750
6	Upgraded P500, or P650
7	P150, or P250-I
8	P850
9	I450-II
10	P550-II
11	P2250
15	P9950
16	P9650
17	P2550
18	P9955
19	P9750
21	P2350
22	P2655
23	P9655
25	P2450
30	P9955-II
34	P9755

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: Load SVCLIB.

DATE$

Purpose

DATE$ returns the current date and time in binary format.

Usage

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

fs_date = DATE$ ();
```

Parameters

fs_date

RETURNED VALUE. Standard FS-format date.

Discussion

DATE$ returns the current date and time in the standard bit-encoded FS format. The FS format for dates is defined 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: No special action.

ERTXT$

This routine accepts a standard PRIMOS error code and returns the character string representation of its error message as it would be printed by the routine ERRPR$.

Usage

```
DCL ERTXT$ ENTRY (FIXED BIN, CHAR(1024)VAR);

CALL ERTXT$ (code, errmsg);
```

Parameters

code

INPUT. Standard error code.

errmsg

OUTPUT. Text of error message.

Discussion

If code is not a valid error code, the null string 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.

GINFO

Purpose

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

Usage

```
DCL GINFO ENTRY ((6) FIXED BIN, FIXED BIN);

CALL GINFO (xervec, n);
```

Parameters

xervec

OUTPUT. Contains n halfwords (up to 6) as follows.

Information for PRIMOS II:

xervec Word	Content
1	Low boundary of PRIMOS II buffers (77777 octal if 64K PRIMOS II).
2	High boundary of PRIMOS II (77777 octal if 64K PRIMOS II).
3	Reserved.
4	Reserved.
5	Low boundary of PRIMOS II and buffer (64K for PRIMOS II only).
6	High boundary of 64K PRIMOS II.

Information for PRIMOS:

xervec Word	Content
1	0
2	0
3-6	Reserved.

n

INPUT. Maximum number of words to return.

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.

PRI$RV

Purpose

This subroutine returns the revision number of the currently running PRIMOS operating system.

Usage

```
DCL PRI$RV ENTRY (CHAR(32)VAR);

CALL PRI$RV (primos_rev);
```

Parameters

primos_rev

OUTPUT. A 32-character varying string containing the PRIMOS revision 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.

RSEGAC$

Purpose

This routine is used to verify that a particular segment exists. It also indicates the requester's access rights to the segment.

Usage

```
DCL RSEGAC$ ENTRY (FIXED BIN(15), FIXED BIN(31)) RETURNS (BIT(1));

seg_exists = RSEGAC$ (segno, access);
```

Parameters

segno

INPUT. The segment number.

access

OUTPUT. The first halfword is reserved.

If the segment exists, the value returned in the second halfword indicates the user's access rights to the segment. Possible values and their interpretations are:

0	No access.
1	Gate Access.
2	Read Access.
3	Read, Write Access.
4,5	Reserved.
6	Read, Execute Access.
7	Read, Write, Execute Access.

seg_exists

OPTIONAL RETURNED VALUE. PL/I true if the segment exists; false if the segment does not exist.

Discussion

If the segment does not exist, the call elicits a return FALSE ('0'b). If the segment exists, a TRUE ('1'b) is returned and the access value for that segment is also returned in the access argument.

FORTRAN programs cannot directly call this subroutine, because it has a seven-character name. A given program may indirectly call it, for example, with CALL SYNYM(SEGNO, ACCESS), and at BIND time rename SYNYM as RSEGAC$.

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.

USER$

Purpose

USER$ returns the user number and user count.

Usage

```
DCL USER$ ENTRY (FIXED BIN, FIXED BIN);

CALL USER$ (current_user_number, user_count);
```

Parameters

current_user_number

OUTPUT. User number of the process issuing the call.

user_count

OUTPUT. Total number of users logged into the system.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: No special action.

USER INFORMATION ROUTINES

This section describes the following subroutines:

Routine	Function
ASSUR$	Checks process has given amount of time slice left.
CHG$PW	Changes login validation password.
COM$AB	Expands a line using abbreviations preprocessor.
IDCHK$	Validates a name.
IN$LO	Determines whether a forced logout is in progress.
LOGO$$	Logs out a user.
PRJID$	Returns the user's project identifier.
PTIME$	Returns amount of CPU time used since login.
PWCHK$	Validates syntax of a password.
READY$	Displays PRIMOS command prompt.
SID$GT	Returns user number of initiating process.
SUSR$	Tests whether current user is supervisor.
TI$MSG	Displays standard message showing times used.
TIMDAT	Returns timing information and user identification.
UNO$GT	Lists users with same name as caller.
UTYPE$	Returns user type of current process.
VALID$	Validates a name against composite identification.

ASSUR$

Purpose

ASSUR$ allows a process to ensure it receives a certain amount of uninterrupted CPU time before its time slice ends.

Usage

```
DCL ASSUR$ ENTRY (FIXED BIN) RETURNS (BIT ALIGNED);

waited = ASSUR$ (desired_time);
```

Parameters

desired_time

INPUT. Time requested, in milliseconds.

waited

OPTIONAL RETURNED VALUE. Set to TRUE ('1'b) if the process waited in a queue before receiving the amount of time requested.

Discussion

ASSUR$ returns immediately if the desired_time is less than the time remaining in the current time slice. ASSUR$ reschedules the process if insufficient time is left in the current time slice.

If desired_time is greater than the time slice, the process obtains only the maximum time slice, and no more.

This procedure should be used when a time-critical application needs to use the CPU uninterrupted by other user processes. Time slices are described in the Operators's Guide to System Commands.

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.

CHG$PW

Purpose

CHG$PW changes the login validation password.

Usage

```
DCL CHG$PW ENTRY (CHAR(16)VAR, CHAR(16)VAR, FIXED BIN);

CALL CHG$PW (old_pw, new_pw, code);
```

Parameters

old_pw

INPUT. The user's current login validation password.

new_pw

INPUT. The new password desired. Passwords may contain any characters except PRIMOS reserved characters (see the Prime User's Guide). Lowercase alphabetic characters are mapped to uppercase by CHG$PW. At the System Administrator's option, null passwords may be disallowed.

code

OUTPUT. Standard error code. Possible values are:

0	Operation succeeded.
E$BPAR	One of the passwords is illegal.
E$BPAS	The old password passed does not match the actual password.
E$EXST	The new password is the same as the old one.
E$WTPR	The disk is write-protected.

Discussion

CHG$PW allows a user to change the login validation password. This is the password that a user gives during the LOGIN command procedure.

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.

COM$AB

Purpose

This procedure expands a line of text using the PRIMOS abbreviation preprocessor.

Usage

```
DCL COM$AB ENTRY (CHAR(*)VAR, FIXED BIN, FIXED BIN);

CALL COM$AB (command, command_size, code);
```

Parameters

command

INPUT/OUTPUT. On input, contains the string to be expanded. On output, contains the expanded string. The input value of command should not be more than 1024 characters long.

command_size

INPUT. Maximum length of command.

code

OUTPUT. Standard error code. Possible values are:

0	Success.
E$TRCL	Expanded line was longer than command_size and was truncated.

Discussion

COM$AB expands command, which can contain any text, as though it were a line typed at the ready prompt. COM$AB displays appropriate error messages if there are problems with the abbreviations file, or the output line is truncated. If abbreviations are turned off, command is not changed. See the Prime User's Guide for more information on the abbreviations preprocessor.

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.

IDCHK$

Purpose

This function checks that the name passed is a legal user or project name. This means that the name must be between 1 and 32 characters long, start with an uppercase letter, and contain only uppercase letters, numbers, and the special characters . (period), $ (dollar sign), and _ (underscore).

Usage

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

id_ok = IDCHK$ (key, id);
```

Parameters

key

INPUT. Restrictions on the name. Keys may be added together:

K$UPRC Mask id to uppercase before checking.

K$WLDC Allow wildcard characters in id. (See the Prime User's Guide.)

K$NULL Allow null ids.

id

INPUT/OUTPUT. The name to check (input unless key is K$UPRC; in that case, input/output).

id_ok

RETURNED VALUE. Set to PL/I true ('1'b) if the name is valid given the restrictions of the keys.

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.

IN$LO

Purpose

This routine is used to determine whether a forced logout is in progress.

Usage

```
DCL IN$LO ENTRY RETURNS (BIT ALIGNED);

in_logout = IN$LO ();
```

Parameters

in_logout

RETURNED VALUE. Returns true ('1'b) if the process has received a forced logout.

Discussion

If the process has an on-unit for the LOGOUT$ condition, it can continue to run for a short time. This function returns true if the process is in this state.

This function cannot be called from FTN because it has no parameters.

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.

LOGO$$

Purpose

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

Usage

```
DCL LOGO$$ ENTRY (FIXED BIN, FIXED BIN, CHAR(32), FIXED BIN,
                  FIXED BIN(31), FIXED BIN);

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

Parameters

key

INPUT. Operation to be performed. Possible values are the following:

-1	Log out all users (supervisor only).
0	Log out self (same as LOGOUT command).
1	Log out specific user by number (same as LOGOUT -NN).
2	Log out specific user by name (supervisor or its phantoms only).

user

INPUT. User number to be logged out. This value is examined only if key is 1.

usrnam

INPUT. Name of user to be logged out; must correspond to number supplied in user. This value is examined only if key is 2.

unlen

INPUT. Length of usrnam in characters. This value is examined only if key is 2.

reserv

Reserved for future use.

code

OUTPUT. Standard error code. Possible values are:

0	No error.
E$BKEY	Bad key.
E$BPAR	Invalid number is specified in user.
E$BNAM	usrnam does not correspond to user.
E$NRIT	Attempt to log out user with name different from 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: No special action.

PRJID$

Purpose

This subroutine is part of the User Registration and Profiles system. It returns the user's project name.

Usage

```
DCL PRJID$ ENTRY (CHAR(32)VAR);

CALL PRJID$ (project_id_name);
```

Parameters

project_id_name

OUTPUT. User's current project name.

Discussion

Trailing blanks on the project name are not returned. If the user is logged into the default project, the returned name is DEFAULT.

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.

PTIME$

Purpose

This procedure reads the amount of CPU time the process has used since login. It is a convenient alternative to TIMDAT if only CPU time is required.

Usage

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

elapsed_time = PTIME$ ();
```

Parameters

elapsed_time

RETURNED VALUE. Indicates the amount of CPU time the process has used since login. The time is returned in units of 1.024 milliseconds.

Discussion

To determine how much CPU time is used during execution of some code sequence, call PTIME$ before the code is executed and save the value; then call PTIME$ after the code is executed. The difference between the values is the time used.

Because this function has no parameters, it cannot be directly called from FTN.

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.

PWCHK$

Purpose

This function makes sure that the password supplied is a legal login password.

Usage

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

pw_ok = PWCHK$ (key, password);
```

Parameters

key

INPUT. An option to restrict values of password. Keys may be added together:

K$UPRC Change password to uppercase before checking.

K$NULL Allow null passwords.

password

INPUT. Must be 1 to 16 characters long, and may not contain PRIMOS reserved characters.

pw_ok

RETURNED VALUE. Set to PL/I true ('1'b) if the password is legal.

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.

READY$

Purpose

READY$ prints a PRIMOS command prompt (the "ready" message).

Usage

```
DCL READY$ ENTRY (BIT(16), FIXED BIN);

CALL READY$ (format, code);
```

Parameters

format

INPUT. Only the most significant bit is used; the rest are reserved. If the most significant bit is 1, the brief form of the prompt is displayed. If the most significant bit is 0, the long form is displayed.

code

INPUT. Error code. If this value is greater than zero, the error prompt is displayed. If the value is less than zero, the warning prompt is displayed. If the value is zero, the normal prompt is displayed.

Discussion

See the Prime User's Guide for a description of the command level prompts. Note that no newline follows the brief forms of the prompts.

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.

SID$GT

Purpose

This procedure returns the user number of the process that started the current process.

Usage

```
DCL SID$GT ENTRY (FIXED BIN);

CALL SID$GT (spawner_id);
```

Parameters

spawner_id

INPUT. User number of the process that started the current process.

Discussion

If the process that calls SID$GT is a phantom, spawner_id is the user number of the user that started the phantom. If the process is a batch job, spawner_id is the user number of the batch server, a special process that manages the batch subsystem.

Interactive users have no spawner. If SID$GT is called by an interactive user, spawner_id 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.

SUSR$

Purpose

SUSR$ determines whether the current process is the supervisor process. This is the process that runs at the operator console.

Usage

```
DCL SUSR$ ENTRY (BIT ALIGNED);

CALL SUSR$ (susr_flag);
```

Parameters

susr_flag

OUTPUT. Returns true ('1'b) if the process is the supervisor process; otherwise returns false ('0'b).

Discussion

In current revisions of PRIMOS, the supervisor user is always User number 1. This will not always be guaranteed to be so, and SUSR$ should be used to test for the supervisor.

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.

TI$MSG

Purpose

TI$MSG types a standard format message that displays elapsed time, CPU time, and I/O time. The standard format is that used by PRIMOS during logout or in response to the TIME command.

Usage

```
DCL TI$MSG ENTRY (FIXED BIN, FIXED BIN(31), FIXED BIN(31),
                  FIXED BIN(31));

CALL TI$MSG (reserv, connect, cpu, io);
```

Parameters

reserv

    INPUT. This value is not used.

connect

    INPUT. Clock time elapsed since login (connect time), in minutes.

cpu

    INPUT. CPU time used, in seconds.

io

    INPUT. I/O time used, in seconds.

Discussion

All the parameters are input parameters. The user must provide the values that the procedure formats and types.

An example of the way this routine can be used is to call LON$R (see Chapter 5) and print the returned values.

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.

TIMDAT

Purpose

TIMDAT returns the date, time, CPU time, and disk I/O time used since login, the user's unique number on the system, and the user-id in a structure.

Usage

```
DCL TIMDAT (1..., FIXED BIN);

CALL TIMDAT (struc, num);
```

Parameters

struc

OUTPUT. A structure of the following elements:

```
2  date char(6),              Current date in MMDDYY format.
2  time,
   3  minutes fixed bin,     Time in minutes since midnight.
   3  seconds fixed bin,     Seconds passed after the minute.
   3  ticks fixed bin,       Ticks passed after the second.
2  CPU_time,
   3  seconds fixed bin,     CPU time used in seconds.
   3  ticks fixed bin,       CPU ticks passed after the second.
2  IO_time,
   3  seconds fixed bin,     Disk I/O time used in seconds.
   3  ticks fixed bin,       Disk I/O ticks passed after the
                             second.
2  ticks_per_sec fixed bin,  Number of ticks per second.
2  user_number fixed bin,    User number.
2  user_name char(32);       User login name.
```

num

INPUT. Indicates maximum number of halfwords to be returned. If this number is more than 28, only 28 halfwords are returned.

Discussion

This routine does not return any useful information under PRIMOS II.

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

FORTRAN programmers should declare the structure as an array of 28 sixteen-bit integers.

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.

UNO$GT

Purpose

This procedure lists all the processes with the same user name as the calling user.

Usage

```
DCL UNO$GT ENTRY ((*)FIXED BIN, FIXED BIN, FIXED BIN);

CALL UNO$GT (id_list, max_ids, num_ids);
```

Parameters

id_list

OUTPUT. An array of 16-bit integers that contains the user numbers of processes that have the same user name as the calling user.

max_ids

INPUT. The maximum length of id_list.

num_ids

OUTPUT. The number of values stored in id_list.

Discussion

If the number of processes with the same name is greater than max_ids, only max_ids values are stored. If this happens, there is no indication of the error.

The calling user's process number is not among those 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.

UTYPE$

Purpose

UTYPE$ returns the user type of the current process.

Usage

```
DCL UTYPE$ ENTRY (FIXED BIN);

CALL UTYPE$ (user_type);
```

Parameters

user_type

OUTPUT. Type of the process making the call. User types are defined below.

Discussion

UTYPE$ returns the user type of the current process. The user type identifies the process by certain classes defined below. It is the preferred method of determining whether or not a given process is a phantom.

These type definitions are inserted into a source by means of the INCLUDE command, as discussed for each language in Volume I. The definitions are provided for FORTRAN, PL/I, and PMA in the following files:

```
SYSCOM>USER_TYPES.INS.FTN
SYSCOM>USER_TYPES.INS.PL1
SYSCOM>USER_TYPES.INS.PMA
```

Users who program in other languages such as Pascal or C should rewrite the SYSCOM file for their languages. The names in this file may not be used in COBOL, as they contain dollar signs. A COBOL program should use the numeric values instead of names.

Possible user types are:

U$NORM	Local terminal user.
U$TREM	User gone to a remote system.
U$FREM	User from a remote system.
U$THRU	User logged through (both to and from remote).
U$SUSR	Supervisor (User 1).
U$TFAM	FAM I running at a user terminal.
U$PH	Cominput-style phantom.
U$CPH	CPL-style phantom.
U$NPX	Slave process.
U$PFAM	FAM I running as a phantom.
U$NET	Network server process (NETMAN).
U$RTS	Route-through server process.
U$FORK	Primix Forked process.
U$LSR	Login Server.
U$LOIP	Logout in progress.
U$BACH	Batch phantom.

Types UNPX, UNET, U$RTS, and U$LSR do not occur in processes that run user programs; they are special process types reserved for use by PRIMOS.

Types U$TFAM and U$PFAM do not occur in new versions of PRIMOS.

There are also four special types that mark the ranges of terminal and nonterminal (phantom) users. These markers are:

U$LTUT	Lowest terminal user type.
U$HTUT	Highest terminal user type.
U$LPUT	Lowest phantom user type.
U$HPUT	Highest phantom user type.

By using these marker types, callers can avoid having to change the range they check when new types are added to the list.

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.

VALID$

Purpose

This routine validates a string against the user's composite identification.

Usage

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

id_valid = VALID$ (name, code);
```

Parameters

name

INPUT. Identification to be checked.

code

OUTPUT. Standard error code. Possible values include:

0	Routine successfully called.
E$BID	name is not a legal identifier. The value of name must be a valid login name or ACL group name.

id_valid

RETURNED VALUE. Set to true ('1'b) if name is either the user's login name or is one of his ACL group names.

Discussion

VALID$ checks an arbitrary string against a combination of the user's login name and ACL groups (the user's composite identification). This routine is used by the File ACL system to determine whether the current user matches some "id:access" pair. The routine is, however, not directly related to the file system and may be of use in another context.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: Not available.

3
User Terminal I/O

This chapter describes procedures that perform input and output on the user's main terminal, as well as procedures for controlling terminal interaction.

The first part of this chapter describes routines used for handling input. For interactive users, input is from the user terminal. By issuing the COMINPUT command (see the Prime User's Guide) or calling the COMI$$ procedure, you can switch input so that it originates from a file. See below for more information on the way the system uses a command input file.

The second part of this chapter describes routines used for handling output. Output is normally to the user terminal, but if the user issues the COMOUTPUT command (see the Prime User's Guide) or calls the COMO$$ procedure, output goes to a file, either exclusively or in addition to the terminal. This section includes a number of routines that are used to build, piece by piece, a line of formatted output. This technique is now obsolete; use IOA$, which is described in this chapter, to perform free-format output.

The third part of this chapter describes routines used to control user terminal I/O.

COMMAND INPUT FILES

There are four situations concerning input from the user terminal:

- If an interactive user starts a program from the terminal, routines accepting input read from the terminal.

- If a command input file is in control and starts a program, most routines accepting input read from the command input file. However, some routines read from the terminal when a command file is in control, giving the programmer the option of reading from the terminal under all circumstances. The individual routine descriptions describe which routines offer this choice. The person writing the command input file must know that the program will be requesting input. If the program attempts to read past the end of the file, the COMI_EOF$ condition is raised.

- If a CPL program is in control and executes a program, the result depends on whether or not a command input file executed the CPL program. If a command input file did execute the CPL program, input is read from the file as in the second case above. If no command input file is in control, input is always taken from the terminal.

- If a CPL program is in control and issues a &DATA command, the lines in the &DATA block are copied to a temporary file, which becomes a command input file. As in the second example, the programmer retains the option of reading from the terminal by choosing the appropriate routines. If the program reads past the end of the temporary command input file, the CPL interpreter catches the COMI_EOF$ condition, issues an appropriate error message, and stops running the CPL file. This event can be avoided by putting the &TTY directive at the end of the &DATA block. The &TTY directive instructs CPL to switch back to the original source of data.

In summary, the program can pick up terminal input in the following ways:

- When run directly by an interactive user

- When run from a CPL program

- When run from a &DATA group within a CPL program, if the &DATA group has a &TTY directive

- By using one of the routines that pick up only terminal input

The program can pick up input from a command input file in the following ways:

- When run from a command input file
- When run from a &DATA group inside a CPL program

The next section discusses the restrictions on phantom process input.

PHANTOM INPUT AND OUTPUT

In this section, information about phantom processes also applies to batch jobs. Phantom processes have no controlling terminal. Attempts to read input from a terminal fail, so phantom processes must read their input from a command input file. Output is discarded unless the user has activated a command output file using the COMOUTPUT command or the COMO$$ routine.

A phantom process may attempt to read from the nonexistent terminal. It might call one of the routines that reads unconditionally from the terminal. It might attempt to read a command input file when no command input file is open. In either case, PRIMOS prints an error message on the supervisor terminal, and logs out the phantom process.

ASSIGNED LINES

This volume only describes character input and output on the user login terminal. Volume IV of the Subroutines Reference Guide describes character input and output on an assigned line. Assigned lines control those terminals and other character-oriented devices not intended for user login.

SINGLE-CHARACTER ARGUMENTS

Some of the routines in this chapter have one or more arguments that are declared as "(2)CHAR". In each case, only the second character is used. The argument can be declared as a 16-bit integer, if this is more convenient for the programmer. If it is, the actual character argument consists of the least significant 8 bits of the integer. This technique is intended to make the routines easy to use from FTN programs.

If the argument is of type INPUT, the first character (or most significant 8 bits of the integer) is ignored. If the argument is of type OUTPUT, the first character is set to 8 zero bits.

The routines of this type are:

C1IN	T1IN
C1IN$	T1OU
C1NE$	ERKL$$

USER TERMINAL INPUT ROUTINES

This section describes the following subroutines:

Routine	Function
C1IN	Reads a character.
C1IN$	Reads a character.
C1NE$	Reads a character, suppressing echo.
CL$GET	Reads a line.
CNIN$	Reads a specified number of characters.
COMANL	Reads a line into a PRIMOS buffer.
RDTK$$	Parses a command line.
T1IB	Reads a character (function).
T1IN	Reads a character (procedure).
TIDEC	Reads a decimal number.
TIHEX	Reads a hexadecimal number.
TIOCT	Reads an octal number.

C1IN

Purpose

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

Usage

```
DCL C1IN ENTRY ((2)CHAR);

CALL C1IN (char);
```

Parameters

char

OUTPUT. Two-byte string into which the character is placed.

Discussion

The next character is read or loaded into char(2), and char(1) is set to all zero bits. If the character is RETURN, char(2) is set to NEWLINE.

If char is declared as a FIXED BIN integer, or the equivalent in other languages, this routine loads the character into the least significant 8 bits of the integer, and sets the most significant 8 bits to zero.

Line feeds are discarded by the operating system and are not read by the C1IN subroutine.

Use C1IN$ or T1IN if there is a requirement to read from the user terminal rather than a command file, even when a command file is active.

If input is from a command input file, and terminal output has not been switched off by the COMO$$ procedure or the COMOUTPUT command, the character is echoed on the terminal. This is the only difference between C1IN and C1NE$. C1NE$ does not echo such characters to the terminal.

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.

C1IN$

Purpose

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

Usage

```
DCL C1IN$ ENTRY ((2)CHAR [, BIT ALIGNED [, BIT ALIGNED ]]);

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

Parameters

char

OUTPUT. Two-byte string into which the character is placed.

echo_flag

OPTIONAL INPUT. If true ('1'b), and input is from a command file, the character is echoed to the terminal. If echo_flag is missing, the assumed value is true.

term_flag

OPTIONAL INPUT. If true ('1'b), input is taken from the terminal regardless of whether or not a command file is active. If term_flag is missing, the assumed value is false.

Discussion

The next character is read into char(2), and char(1) is set to all zero bits. If the character typed is RETURN, char(2) is set to NEWLINE.

Calling C1IN$ with echo_flag and term_flag omitted is equivalent to calling C1IN (see previous description).

In V-mode and I-mode, calling C1IN$ with term_flag true is equivalent to calling T1IN (see later in this chapter). However, C1IN$ is implemented more efficiently than T1IN. Use C1IN$ in preference to T1IN if efficiency is more important than the slightly more complicated calling sequence of C1IN$.

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.

C1NE$

Purpose

This routine gets the next character either from the terminal or from a command file, depending upon the command stream source. If a command input file is active, the character is not echoed to the terminal.

Usage

```
DCL C1NE$ ENTRY ((2)CHAR);

CALL C1NE$ (char);
```

Parameters

char

OUTPUT. Two-byte string into which the character is placed.

Discussion

The next character is read or loaded into char(2), and char(1) is set to all zero bits. If the character is RETURN, char(2) is set to NEWLINE.

If input is from a command input file, the character is not echoed to the terminal. This is the only difference between C1NE$ and C1IN. C1IN does echo all such characters to the terminal.

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$GET

Purpose

CL$GET reads a single line of input text from the currently defined command input stream (terminal or command file).

Usage

```
DCL CL$GET ENTRY (CHARACTER(*)VARYING, FIXED BIN, FIXED BIN)
                 RETURNS (FIXED BIN);

comi_switch = CL$GET (comline, comline_size, code);
```

Parameters

comline

OUTPUT. Varying character string into which the text is read from the command input stream. Because comline is of type character varying, no blanks or zeroes are added beyond the last character read.

comline_size

INPUT. Maximum length (in characters) of comline.

code

OUTPUT. Standard error code.

comi_switch

OPTIONAL RETURNED VALUE. Zero if input was read from the user terminal, and nonzero if input was read from a file.

Discussion

The line is returned as a varying character string without the NEWLINE character at the end. An empty command line returns the null string, but one consisting of all blanks is handled as a command line containing ordinary characters.

The user's erase and kill characters are processed by CL$GET. CL$GET is preferable to CNIN$ for most purposes. Most applications programs do not perform their own erase and kill processing.

Example

Below is an example using the subroutine CL$GET.

```
OK, SLIST CLGET1.PASCAL

{<readtty.pascal> Reads text from the user terminal using the external}
{                 PRIMOS routine CL$GET                               }
{                                                                     }
{This program provides an example of how to implement the Pascal      }
{equivalent of the character varying datatype found in PL/I.  The     }
{Prime Pascal extension STRING data type has the same structure       }
{as the CHARACTER VARYING type.  The default length of a STRING       }
{variable is 80.  The Prime extension STRING functions LENGTH and     }
{SUBSTR are identical to the PL/I functions of the same names.        }
{                                                                     }
{The simple object of the program is to read three strings from the   }
{terminal and display them in complete reverse order.                 }
{                                                                     }
program readTTY;

type
  char80varying = string;      {Can also be declared as string[80]}

var
  cmdline : char80varying;
  table   : array[1..3] of char80varying;
  i, j    : integer;
  code    : integer;

procedure cl$get(var cmdline: char80varying;{Command line input buffer}
                     lenbytes: integer;     {Length of cmdline in bytes}
                 var code    : integer);     {Return error code status }
          extern;                           {External PRIMOS procedure}

begin
  {Loop to input the text entered from the user terminal using the    }
  {PRIMOS routine defined above (cl$get).                             }
  {                                                                   }
  for i := 1 to 3 do
    begin
      write(i:1,'> ');
      cl$get(cmdline, 80, code);
      if code <> 0 then
        writeln('Bad status code returned, status =',code);
    table[i] := cmdline;                        {Save the command line}
  end;
  writeln;
  {                      Display the lines just typed in reverse order}
```

```
  for i := 3 downto 1 do
    begin
      write(i:1,'< ');
      for j := length(table[i]) downto 1 do
        write(substr(table[i], j, 1));
      writeln;
    end;
end.
```

This program, stored as CLGET1.PASCAL, can be compiled, loaded, and run with the following dialog:

```
OK, PASCAL CLGET1
[PASCAL Rev. 20.2 Copyright (c) 1986, Prime Computer, Inc.]
0000 ERRORS [PASCAL Rev. 20.2]
OK, BIND
[BIND Rev. 20.2 Copyright (c) 1985, Prime Computer, Inc.]
: LO CLGET1
: LI PASLIB
: LI
BIND COMPLETE
: FILE
OK, RESUME CLGET1
1> ABCDE
2> SECOND
3> MADAMIMADAM

3< MADAMIMADAM
2< DNOCES
1< EDCBA
OK,
```

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.

CNIN$

Purpose

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

Usage

```
DCL CNIN$ ENTRY (CHARACTER(*), FIXED BIN, FIXED BIN);

CALL CNIN$ (buffer, char_count, actual_count);
```

Parameters

buffer

OUTPUT. A buffer in which the string of characters read from the input stream is to be placed.

char_count

INPUT. The number of characters to be transferred from the input stream to buffer.

actual_count

OUTPUT. A returned argument. It specifies the number of characters read by the call to CNIN$. If reading continues until a NEWLINE character is encountered, the count includes the NEWLINE character.

Discussion

CNIN$ reads from the input stream until either a NEWLINE character is encountered or the number of characters specified by char_count is read. If an odd number of characters is read, the remaining character space in the last halfword is not modified. The erase and kill characters are not interpreted.

Input to CNIN$ is obtained from the terminal unless the process is reading from a CPL &DATA block, or the user has previously given the COMINPUT command, and this command is still in control. The COMINPUT and &DATA commands switch the input stream so that it comes from a file

rather than from the terminal. A phantom can only read commands from its command file. (Refer to the Prime User's Guide for further information.)

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.

COMANL

Purpose

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

Usage

```
DCL COMANL ENTRY;

CALL COMANL;
```

Parameters

There are no parameters.

Discussion

The line is read into an internal text buffer. This buffer is internal to PRIMOS and can be accessed only by a call to RDTK$$. The buffer holds 80 characters.

Use of COMANL and RDTK$$ to read parameters is obsolete in PL/I and Pascal. The preferred method is to use CL$GET and CL$PIX.

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.

RDTK$$

Purpose

RDTK$$ parses the command line most recently read by a call to COMANL. If no previous calls to COMANL have taken place, RDTK$$ parses the last command line typed at PRIMOS command level by the user. RDTK$$ is obsolete; CL$PIX should be used instead.

Usage

```
DCL RDTK$$ ENTRY (FIXED BIN, (8) FIXED BIN, CHAR(*), FIXED BIN,
                  FIXED BIN);

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

Parameters

key

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

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

info

OUTPUT. An eight-halfword integer array set to contain the following information (only info(2) is set for a key value 4):

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

1	Normal token. (Results of numeric conversions are returned.)
2	Register setting parameter.
5	Null token.
6	End of line.

info(2) The length in characters of the token. A null token has a 0 length.

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

bit 1 (:100000) -- Decimal conversion successful (no overflow), value returned in info(4).

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

bit 3 (:020000) -- Token begins with unquoted minus sign, thus token can be a keyword argument.

bit 4 (:010000) -- An explicit position for a register setting was given; position value is returned in info(4).

bits 5-16 Reserved.

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

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

info(6)-(8) Reserved.

buffer

OUTPUT. A character string into which the literal text of the token is written by RDTK$$ and blank-padded to length buflen, in halfwords.

buflen

INPUT. The specified length (in halfwords) of buffer. buflen must be >= 0.

code

OUTPUT. Standard error code. Possible values are:

0	No errors.
E$BKEY	Value of key is illegal.
E$BPAR	Bad parameter; buflen is less than 0.
E$BFTS	Value of buflen is too small to contain the full text of the token. The token is truncated.

Discussion

RDTK$$ is obsolete. CL$PIX should be used instead for parsing lines read using CL$GET. CL$PIX should also be used for parsing the command lines of EPF (Executable Program Format) programs. For other cases, you can recover the whole line with RDTK$$, using key value 4, convert it to type character varying, and analyze it using CL$PIX.

Parsing proceeds token by token. A command line consists of tokens (for definitions, see Tokens section later in this chapter) separated by delimiters. The current delimiters are:

```
space  comma  /*  NEWLINE
```

The reserved characters in command lines are:

```
(  [  {  )  ]  }  !  ;  ^  "  ?  :  ~  |  \  .DEL.
```

However, you can include one of these characters in a token by enclosing the token in single quotes; for example, 'awful(so to speak)'. The /* characters, if unquoted, begin a comment field that extends to the end of the line and are ignored by RDTK$$.

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

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

SP , ' NL /*

The meanings of these characters are discussed in the next paragraphs.

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

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

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

'HERE''S A LITERAL ''.'

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

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

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

Tokens

A token is any string of characters not containing a delimiter. A token can be from 0 to 80 characters in length. The following are examples of valid tokens:

```
FTN
LONG-FILENAME
1/707
6
98
String.even.longer.than.thirty-two.characters
[path]name
.NULL. (null string)
```

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

```
'STRING WITH EMBEDDED BLANKS'
'HERE''S A LITERAL QUOTE MARK'
```

Token Types

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

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

```
FTN
A0001
This.is.a.token.
PARTIALLY' L I T E R A L'
'8'xxx  (Note: '8' is treated as a nonnumeric.)
''''''''      (= '''')
```

Register-setting Token: Register-setting tokens, or octal parameters (explained in the LOAD and SEG Guide), are now considered obsolete. They are handled by RDTK$$ solely to permit existing software and command files to continue to function. New software should not use such parameters; symbolic keywords should be used instead: for example, FTN XX -64V instead of FTN XX 2/400.

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

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

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

X,,Y
```

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

Strategy

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

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

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

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

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

A buflen of 0 can be used to skip over a token. The error code E$BFTS is returned.

For a key of 4 (read raw text), all text between the current RDTK$$ pointer and the end of the command line (NEWLINE) is returned. No checking is done for any delimiters or special characters other than NEWLINE. No forcing to uppercase is performed.

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.

T1IB

Purpose

T1IB reads one character from the user terminal.

Usage

```
DCL T1IB ENTRY RETURNS (FIXED BIN);

charval = T1IB ();
```

Parameters

charval

RETURNED VALUE. Input character.

Discussion

charval contains the binary equivalent of the character just read. charval must be declared as a 16-bit integer, not as a character string.

This function always reads from the terminal. Use C1IN if there is a requirement to read a character from an active command input file.

This function cannot be called from FTN, as it has no parameters. Use C1IN$ or T1IN instead.

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.

T1IN

Purpose

T1IN reads one character from the user terminal.

Usage

```
DCL T1IN ENTRY ((2)CHAR);

CALL T1IN (char);
```

Parameters

char

OUTPUT. Two-byte string into which the character is placed.

Discussion

The next character is read or loaded into char(2), and char(1) is set to all zero bits. If a RETURN is read, a NEWLINE is output and char is set to NEWLINE. If a LINEFEED (NEWLINE) character is read, it is discarded by PRIMOS.

If char is declared as a FIXED BIN integer, or the equivalent in other languages, this routine loads the character into the least significant 8 bits of the integer, and sets the most significant 8 bits to zero.

Use C1IN if there is a requirement to read from an active command file.

The routine C1IN$ (described earlier in this chapter) is also capable of forcing input to come from the terminal, and is implemented more efficiently than T1IN. Use C1IN$ in preference to T1IN if efficiency is more important than the slightly more complicated calling sequence of C1IN$.

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.

TIDEC

Purpose

TIDEC reads terminal input as a decimal number.

Usage

```
DCL TIDEC ENTRY (FIXED BIN);

CALL TIDEC (variable);
```

Parameters

variable

OUTPUT. Binary value of character string typed.

Discussion

The number may be preceded by a minus sign to indicate that it is negative, but must not be preceded by a plus sign. Numbers can be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is invalid, and more input is then accepted. A space or carriage return is then accepted as a zero.

This routine does not carry out erase or kill processing.

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.

TIHEX

Purpose

TIHEX reads terminal input as a hexadecimal number.

Usage

```
DCL TIHEX ENTRY (FIXED BIN);

CALL TIHEX (variable);
```

Parameters

variable

OUTPUT. Binary value of character string typed.

Discussion

The number may be preceded by a minus sign to indicate that it is negative, but must not be preceded by a plus sign. Numbers can be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is invalid, and more input is then accepted. A space or carriage return is then accepted as a zero.

This routine does not carry out erase or kill processing.

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.

TIOCT

Purpose

TIOCT reads terminal input as an octal number.

Usage

```
DCL TIOCT ENTRY (FIXED BIN);

CALL TIOCT (variable);
```

Parameters

variable

OUTPUT. Binary value of character string typed.

Discussion

The number may be preceded by a minus sign to indicate that it is negative, but must not be preceded by a plus sign. Numbers can be terminated by a carriage return or a space. A question mark or other error message is displayed if a numeric input is invalid, and more input is then accepted. A space or carriage return is then accepted as a zero.

This routine does not carry out erase or kill processing.

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.

USER TERMINAL OUTPUT ROUTINES

This section describes the following subroutines:

Routine	Function
ERRPR$	Prints a standard error message.
IOA$	Provides free-format output.
IOA$ER	Provides free-format output, for error messages.
TNOU	Writes characters to terminal, followed by NEWLINE.
TNOUA	Writes characters to terminal.
TODEC	Writes a signed decimal number.
TOHEX	Writes a hexadecimal number.
TONL	Writes a NEWLINE.
TOOCT	Writes an octal number.
TOVFD$	Writes a decimal number, without spaces.
T1OB	Writes one character from Register A.
T1OU	Writes one character.

ERRPR$

Purpose

ERRPR$ interprets a return code and, if the code is nonzero, prints a standard message associated with the code, followed by optional user text. See Volume I of the Subroutines Reference Guide for more details on error handling.

Usage

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

CALL ERRPR$ (key, code, text, textlen, filnam, namlen);
```

Parameters

key

INPUT. The action to take after printing the message. Possible values are:

K$NRTN	Exit to the system; the system cannot return to the calling program.
K$SRTN	Exit to the system; return to the calling program following a START command.
K$IRTN	Return immediately to the calling program.

code

INPUT. A variable containing the return code from the routine that generated the error. If code is 0, ERRPR$ always returns immediately to the calling program and prints nothing.

text

INPUT. A message to be printed following the standard error message. Text is omitted by specifying textlen as 0.

textlen

INPUT. The length (in characters) of text.

filnam

INPUT. The name of the program or subsystem detecting or reporting the error. filnam is omitted by specifying namlen as 0.

namlen

INPUT. The length (in characters) of filnam.

Discussion

If ERRPR$ is called from an EPF (executable program format) program, using one of the key values K$NRTN, or K$SRTN signals a condition. A key of K$NRTN causes the condition STOP$ to be signalled, with return prohibited. By default, the STOP$ condition returns control to the current command level. A key of K$SRTN causes the condition R0_ERR$ to be signalled, with return permitted. By default, the R0_ERR$ condition generates a new 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: No special action.

IOA$

Purpose

IOA$ provides free-format terminal output.

Usage

```
CALL IOA$ (control, conlen [, arg1, ... argn ]);
```

There is no DCL statement because IOA$ can be called at different times with different numbers and types of arguments. See Note below.

Parameters

control

INPUT. Template string (CHARACTER NONVARYING). See Discussion below for the format of this string.

conlen

INPUT. Length of control (FIXED BIN). If control is self-terminating, conlen may be larger than the active length of control. For more information, see Discussion below.

arg1, ... argn

INPUT. Data for variable fields in string. There may be between zero and 99 data arguments.

IOA$ is designed so that different calls can have a different number of parameters and the parameters can have any data type. If IOA$ is called from PL/I, each PL/I procedure must declare IOA$ with the parameters and types specified, and the module can only make calls to IOA$ with those parameter types. These comments also apply to Pascal.

In FTN, F77, and COBOL, IOA$ can be called with varying numbers of parameters in different places. The CBL compiler issues a warning message, which may be ignored.

C programmers should use the standard C procedure printf, on which IOA$ is based. Pascal programmers should use write, a standard Pascal procedure.

Discussion

IOA$ provides free-format output to the terminal. In general, application programs use the standard output package provided with the programming language. Systems programs can benefit from the efficiency of IOA$. Another advantage of IOA$ is that it provides more flexibility than do most language support packages. Also, the format of the IOA$ template is simple, and can even be constructed at runtime.

The first parameter, control, is a string that provides a template for the output. The string contains a mixture of text and control codes; control codes are introduced by a character pair made up of the escape character and the percent symbol.

Any character not in a control code is output to the terminal. Most control codes cause data to be formatted and written onto the terminal. The data to be formatted is taken from the variable-length argument list. IOA$ maintains an internal pointer that initially points to arg1. When a control sequence calls for the next argument, IOA$ uses the argument currently pointed to and advances the pointer. If IOA$ runs out of arguments, output stops immediately. If IOA$ reaches the end of control without using all the arguments, the excess arguments are ignored.

You must ensure a match between the control codes and the actual arguments. IOA$ cannot detect an attempt to convert a parameter of an inappropriate type.

Here is a simple example. The statement below converts the value of the 16-bit integer variable code to characters, and types the string with the value inserted:

```
CALL IOA$('CODE IS %D.', 11, code);
```

The resulting string may look like this:

```
CODE IS 99.
```

Control Code Format

The format of a control code sequence is as follows:

```
%fw:precZRtype
```

The notations fw, prec, and type each stand for a single character or possibly (in the case of fw) a sequence of characters. Only the %

(percent symbol) and type are required; the other parts are optional. The parts of the code are:

fw — Field width, or (occasionally) repeat count. This is normally an integer, but may be a # character (number sign). If the conversion uses this as a field width, the output data occupies this number of characters. If the specified field width is zero, the output data occupies as much space as is necessary to contain it. If the data needs fewer than fw characters, the data is justified either right or left, as noted with the individual type descriptions below.

If fw is negative or omitted, it assumes the value of 0 for a field width, or 1 for a repeat count.

If fw is the character # instead of an integer, the actual field width (or repeat count) is taken from the next argument, which is interpreted as a halfword integer.

:prec — Precision. Note the required colon. This refers to numeric fields, and indicates the type of integer provided as an argument. Possible values for prec are:

0 Unsigned 16-bit integer
1 Signed 16-bit integer
2 Signed 32-bit integer
3 Unsigned 32-bit integer

If the precision specifier is omitted, the default value is 1.

Z — If the letter Z is present, an integer is zero-filled to the field width, rather than space-filled. Z may be in either uppercase or lowercase. The X and L conversions use Z in a special way; see the descriptions of these conversions, below.

R — If the letter R is present, the normal sense of justification is reversed. Fields normally left-justified will be right-justified, and vice versa. R may be in either uppercase or lowercase.

type — A character indicating the type of conversion to be applied. If the character is a letter, the letter may be in either uppercase or lowercase.

The type characters, and the conversions they represent, are as follows:

% Output a single % (percent symbol) to the terminal. The field width, precision, Z, and R, are ignored.

D Output the next argument as a decimal number, right-justified. If the field width is too small to contain the number, as many characters as needed are output.

O Same as D above, except the number is output in octal.

H Same as D above, except the number is output in hexadecimal.

W Octal halfword. %W is equivalent to %:0ZO.

C Character string. The next argument is the string (nonvarying), and the argument after that is a halfword integer giving the string's length. If the length is negative, it is treated as zero. The string is left-justified. Precision and Z are ignored.

A Trimmed character string. Same as C, except the specified string length is adjusted downwards by removing trailing blanks from the string.

V Varying character string. The next argument is a string of type character varying. It is displayed left-justified. Precision and Z are ignored.

L Logical. The next argument is a 16-bit integer (precision is ignored) that is regarded as true if any bit is 1. If Z is not present, the result of the conversion is the letter T or F. If Z is present, the result is the word TRUE or FALSE. The output is right-justified.

P Pointer. The next argument is a pointer that can be 2 or 3 words long. The pointer's value is displayed in the standard Prime format, and is left-justified. Precision and Z are ignored.

X Output fw filler characters. The filler is 0 (zero) if Z is present; normally it is the space character. Precision and R are ignored.

/ Output fw newlines. Precision, Z, and R are ignored.

^ Output fw form feed characters. Precision, Z, and R are ignored.

$ Terminate control string immediately. If the string ends with %$, you do not need to count the characters in control; conlen can be any number equal to or greater than the actual string length.

Terminate control immediately (as with %$) and output a newline.

(Start repeat group. The repeat count is fw, which must be nonzero. All text and conversions between the %(and the next %) are repeated fw times. Precision, Z, and R are ignored. The repeat group should not contain a nested %(string.

) End repeat group (see above).

Y Reposition in the argument list. The fw value indicates where to reposition; a value of 1 (or less) repositions to the first of the variable arguments. A value of greater than 99 is treated as 99.

If a conversion specifier does not follow the format rules, the result is undefined.

Examples

Two examples are supplied below: the first is in FTN and the second is in PL/I.

The following FTN subroutine accepts two arguments: a string and the string's length. It displays the string and its length, followed by the string's address:

```
      SUBROUTINE DISP(ISTR, ILEN)
      CALL IOA$('"%c" has %d characters.%.', 100,
     1   ISTR, ILEN, ILEN)
      CALL IOA$('It is at %p%.', 100, LOC(ISTR))
      RETURN
      END
```

If the following call is made:

```
      CALL DISP('TEST STRING', 11)
```

the output is:

```
"TEST STRING" has 11 characters.
It is at 4335(3)/1001
```

The following PL/I subroutine has two arguments: a string and a 32-bit integer. It first displays the string in a 20-column field, indented by 4 spaces, and then displays the number in hexadecimal.

```
disp2: proc(string, value);

declare string char(*)varying,
        value fixed bin(31),
        ioa$ entry (char(*), bin, char(*)var, bin(31));

call ioa$('%4x%20v%8:2zh%.', 100, string, value);

end;
```

If the following call is made:

```
call disp2('Hexadecimal value:', 12345678);
```

the output is:

```
    Hexadecimal Vvalue:  00BC614E
```

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.

IOA$ER

Purpose

IOA$ER provides free-format terminal output. Its most frequent use is for displaying error messages, because it forces terminal output on.

Usage

```
CALL IOA$ER (control, conlen, arg1, ... argn);
```

There is no DCL statement because IOA$ER can be called at different times with different numbers and types of arguments. More information is given in the IOA$ description.

Parameters

control

INPUT. Template string (CHARACTER NONVARYING). See the Discussion section of IOA$ for the format of this string.

conlen

INPUT. Length of control (FIXED BIN). If control is self-terminating, conlen may be larger than the active length of control. See the Discussion section of IOA$ for more information.

arg1, ... argn

INPUT. Data for variable fields in string. There may be between zero and 99 data arguments. If there are more than 99 arguments, the excess arguments are ignored.

Discussion

IOA$ER differs from IOA$ in one respect. Before the text is output to the terminal, command output is forced on. This ensures the user will see the message, even if command output has been turned off by the COMOUTPUT command or the COMO$$ procedure.

See the description of IOA$ for further discussion of the meaning of the parameters.

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.

TNOU

Purpose

TNOU writes a specified number of characters to the user terminal followed by a line feed and carriage return.

Usage

```
DCL TNOU ENTRY (CHAR(*), FIXED BIN);

CALL TNOU (buffer, count);
```

Parameters

buffer

INPUT. Text to be written.

count

INPUT. Number of characters to be written.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: No special action.

TNOUA

Purpose

TNOUA writes a specified number of characters to the user terminal, without appending a line feed or carriage return.

Usage

```
DCL TNOUA ENTRY (CHAR(*), FIXED BIN);

CALL TNOUA (buffer, count);
```

Parameters

buffer

    INPUT. Text to be written.

count

    INPUT. Number of characters to be written.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: No special action.

TODEC

Purpose

TODEC outputs a six-character signed decimal number.

Usage

```
DCL TODEC ENTRY (FIXED BIN);

CALL TODEC (variable);
```

Parameters

variable

INPUT. Value of number to be typed.

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.

TOHEX

Purpose

TOHEX outputs a four-character unsigned hexadecimal number.

Usage

```
DCL TOHEX ENTRY (FIXED BIN);

CALL TOHEX (variable);
```

Parameters

variable

INPUT. Value of number to be typed.

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.

TONL

Purpose

TONL outputs a carriage return and line feed.

Usage

```
DCL TONL ENTRY;

CALL TONL;
```

Parameters

There are no parameters.

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.

TOOCT

Purpose

TOOCT outputs a six-character unsigned octal number.

Usage

```
DCL TOOCT ENTRY (FIXED BIN);
CALL TOOCT (variable);
```

Parameters

variable

INPUT. Value of number to be typed.

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.

TOVFD$

Purpose

TOVFD$ writes a 16-bit integer to the terminal.

Usage

```
DCL TOVFD$ ENTRY (FIXED BIN);

CALL TOVFD$ (variable);
```

Parameters

variable

INPUT. Value of number to be typed.

Discussion

This subroutine writes number, which should be a 16-bit integer, to the terminal without any spaces (for example, 123 or -17).

Loading and Linking Information

V-mode and I-mode: No special action to load.
Link with FORTRAN_IO_LIBRARY.

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

R-mode: No special action.

T1OB

Purpose

T1OB writes one character from Register A to the user terminal. This procedure can be called only from PMA, because the user must have access to Register A.

Usage

```
CALL T1OB;
```

No DCL statement is provided because the routine can only be called from PMA.

Parameters

There are no parameters.

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.

T1OU

Purpose

T1OU writes a character to the user terminal.

Usage

```
DCL T1OU ENTRY ((2)CHAR);

CALL T1OU (char);
```

Parameters

char

INPUT. The character in char(2) is typed.

Discussion

If the data type of char is a 16-bit integer, the least significant 8 bits of the integer form the character to be typed.

If char is NEWLINE, RETURN and NEWLINE are output to the user terminal.

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.

USER TERMINAL CONTROL ROUTINES

This section describes the following subroutines:

Routine	Function
BREAK$	Inhibits or enables BREAK function.
CO$GET	Returns information about command output settings.
COMI$$	Switches input between the terminal and a file.
COMO$$	Switches output between the terminal and a file.
DUPLX$	Controls the way PRIMOS treats the user terminal.
ERKL$$	Reads or sets the erase and kill characters.
QUIT$	Determines if there are pending quits.
TTY$IN	Checks for unread terminal input characters.
TTY$RS	Clears the terminal input and output buffers.

BREAK$

Purpose

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

Usage

```
DCL BREAK$ ENTRY (FIXED BIN);

CALL BREAK$ (logic_value);
```

Parameters

logic_value

INPUT. A 16-bit integer whose value can be 1 (TRUE) or 0 (FALSE).

Discussion

The LOGIN command initializes the user terminal so that the CONTROL-P or BREAK key causes an interrupt (QUIT). The BREAK$ routine, if called with the argument 0, enables the CONTROL-P or BREAK key to interrupt a running program.

The BREAK$ routine called with the argument 1 inhibits the CONTROL-P or BREAK characters from interrupting a running program.

This routine maintains a master list of the QUIT status for each user. Each call to BREAK$, to inhibit or enable QUIT, increments or decrement a counter, respectively. QUITs are enabled only when the counter is 0; the counter becomes positive with inhibit requests, and cannot be decremented below 0.

While QUITs are inhibited, the user can still determine if a CONTROL-P was typed by using the QUIT$ routine.

BREAK$ has no effect under PRIMOS II.

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.

CO$GET

Purpose

CO$GET retrieves information about the state of the user's command output (COMO) settings.

Usage

```
DCL CO$GET ENTRY (FIXED BIN, FIXED BIN);

CALL CO$GET (reserved, status);
```

Parameters

reserved

OUTPUT. Reserved.

status

OUTPUT. The least significant two bits of this halfword indicate the state of the command output stream. The bit settings are independent of each other. The meanings are as follows:

Bit number	Meaning
1	If set (1), command output will go to the terminal. If clear (0), the terminal will receive no command output.
2	If set (1), a command output file is active. If clear (0), there is no active command file, or a command file is active and paused.

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.

COMI$$

Purpose

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

Usage

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

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

Parameters

filnam

INPUT. The name of the command file to receive the command input stream (integer array). If filnam begins with the string TTY, the command stream is switched back to the terminal and funit is closed. If filnam begins with the string PAUSE, the command stream is switched to the terminal but the file unit specified by funit is not closed. If filnam begins with the string CONTIN, the command stream is switched to the file already open on funit. Strings beginning with TTY, PAUSE, or CONTIN cannot be used as filenames.

namlen

INPUT. The length (in characters) of filnam.

funit

INPUT. The file unit on which to open the command file specified by filnam. Normally, file unit 6 is used.

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: No special action.

COMO$$

Purpose

COMO$$ switches terminal output to a file or terminal.

Usage

```
DCL COMO$$ ENTRY (BIT(16), CHAR(*), FIXED BIN, FIXED BIN,  FIXED  BIN);

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

Parameters

key

INPUT. A halfword of flags specifying the action to be taken. The values below are specified in octal:

:000001 Turn TTY output off.

:000002 Turn TTY output on.

:000010 Turn file output off.

:000020 Turn file output on.

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

:000100 Truncate filnam if filnam is being opened.

filnam

INPUT. The name of the file to be opened.

namlen

INPUT. The length (in characters) of filnam.

xx

INPUT. Reserved. Should be specified as 0.

code

OUTPUT. Standard error code from the file system.

Discussion

Routing of the terminal output stream is modified as indicated by the key. If TTY output is turned off, all printing at the terminal is suppressed until TTY output is reenabled or until a command output file error message is generated. If a filename is specified, any current command output file is closed, and then the new file is opened for writing. All subsequent terminal output is sent to the new file. TTY output continues unless explicitly suppressed. Unless the APPEND option bit is set, the current contents of the file are overwritten. The parameter can be omitted by specifying a pair of blanks or a length of 0.

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

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.

DUPLX$

Purpose

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

Usage

```
DCL DUPLX$ ENTRY (BIT(16)) RETURNS (BIT(16));

old_tcw = DUPLX$ (tcw);
```

Parameters

tcw

INPUT. Terminal configuration word. See below.

old_tcw

OPTIONAL RETURNED VALUE. Both tcw and old_tcw represent the terminal configuration word, which is a 16-bit integer whose bits have the following meanings (the values below are specified in octal):

Bit	Mask	Meaning If Bit Is SET
1	:100000	Half duplex.
2	:040000	Do not echo LINEFEED after CARRIAGE RETURN.
3	:020000	Turn on XOFF/XON character recognition.
4	:010000	Output currently suppressed (XOFF received).
5	:004000	Detect DATA SET BUSY before output to AMLC line. (See AMLC Functions below.)

Bit	Mask	Meaning If Bit Is SET
6	:002000	Handle reverse channel functionality. (See AMLC Functions below.)

Data Set	Sense Bits	
	Bit 6=1	Bit 6=0
1 (off)	XOFF	XON
0 (on)	XON	XOFF

7	:001000	Check for certain error conditions: o Overflow of the input buffer o Parity error If one of these conditions is present, the character found is replaced with the NAK character.
8	:000400	Indicates a parity error (output). Overflow of the input buffer is flagged when there is only room for one more character.
9-16	:000377	Internal buffer number (read-only).

Discussion

DUPLX$ has no effect under PRIMOS II.

DUPLX$ returns the terminal configuration word and internal buffer number as the value of the function. DUPLX$ must be declared as a function if the returned value is to be used by the calling program.

If the terminal configuration word passed to DUPLX$ is set to all ones, no updating of the configuration word takes place, and the current value is returned.

The tcw of a user terminal is not affected by the LOGIN or LOGOUT commands. The tcw of the user terminal can also be set at the supervisor terminal by using the SET_ASYNC command or the AMLC command. Users can also use the PRIMOS command TERM to change their terminal characteristics.

AMLC Functions

Certain devices require a reverse channel protocol to signal BUSY or READY. For these cases, the carrier detect line is used for the signal. Bit 5 of the terminal configuration word instructs the AMLC (Asynchronous Multi-line Controller) software to interrogate the carrier signal for that line before writing to the device. If a BUSY is detected, then the AMLC software simulates an XOFF received for that line. When the carrier signal goes to the READY state, the AMLC software flags it as an XON, and output resumes. For example, if the device signals BUSY as DATA SET OFF (1), then the terminal configuration word bit setting is:

Bit 5=1 Detect DATA SET sense.

Bit 6=1 If DATA SET sense is off, then simulate XOFF;
else set XON.)

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.

ERKL$$

Purpose

The ERKL$$ subroutine reads or sets the user's definitions of the erase and kill characters.

Usage

```
DCL ERKL$$ ENTRY (FIXED BIN, (2)CHAR, (2)CHAR, FIXED BIN);

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

Parameters

key

INPUT. The action to be taken. Possible values are:

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

erase

INPUT or OUTPUT. With key K$WRIT, the character contained in erase(2) replaces the user's erase character. If erase(2) contains all zero bits, no action takes place. On key K$READ, the user's erase character is placed in erase(2).

kill

INPUT or OUTPUT. With key K$WRIT, the character contained in kill(2) replaces the user's kill character. If kill(2) contains all zero bits, no action takes place. On key K$READ, the user's kill character is placed in kill(2).

code

OUTPUT. Standard error code. Possible values are:

0	No errors.
E$BKEY	Invalid value for key.
E$BPAR	Attempt to set the erase and kill characters to the same value.

Discussion

Erase and kill characters are interpreted by commands to the operating system and by most of the subroutines that perform terminal input. Exceptions are noted with the subroutine description. All language processors' I/O facilities are affected.

Note

RDASC, I$AA12, and I$AA01 are library subroutines that read the user's erase and kill characters only once, when they are first invoked. Changing the erase and kill characters after a call to those subroutines does not affect erase and kill processing in these subroutines until the next program is invoked. The main purpose for users calling the ERKL$$ subroutine is to read or set these characters when the user programs do their own erase and kill processing.

Under PRIMOS II, the erase and kill characters can be read but any attempt to set them is ignored.

The erase and kill characters can be set at command level by the PRIMOS TERM command. The characters are reset to default values upon an explicit logout or login.

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.

QUIT$

Purpose

QUIT$ determines if there are pending terminal quits, and removes the record of them. QUIT$ reads, and then clears, the bit that recorded that a CONTROL-P was typed.

Usage

```
DCL QUIT$ ENTRY(FIXED BIN);

CALL QUIT$ (pending);
```

Parameters

pending

OUTPUT. Set to 0 if there are no quits pending. Set to 1 if there is a quit pending.

Discussion

Recognition of terminal quits may be deferred if the user calls BREAK$. If recognition of quits is deferred, and a CONTROL-P has been typed, QUIT$ returns a value of 1 in pending. If recognition of quits is not deferred, QUIT$ always returns a value of 0 in pending.

QUIT$ also removes the pending quits. You may use BREAK$ and QUIT$ together as a simple way of servicing quit requests without having to use the condition mechanism.

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.

TTY$IN

Purpose

This function checks whether there are any characters in the user's TTY input buffer. The state of the buffer is undisturbed by the call; no character is actually read or removed from the buffer.

Usage

```
DCL TTY$IN ENTRY () RETURNS (BIT(1)ALIGNED);

more_to_read = TTY$IN ();
```

Parameters

more_to_read

RETURNED VALUE. True ('1'b) if there is at least one character of input available at the terminal of the calling process, and '0'b otherwise.

Discussion

TTY$IN is used to check if the user has typed at least one character that has not yet been read by the process. TTY$IN allows the program to poll for input and perform other processing while waiting for the input to arrive. All terminal input routines wait for a character to be typed before returning to the caller.

If TTY$IN is called in a noninteractive process, '0'b is always returned, whether or not a command input file is active.

It is possible for TTY$IN to return '1'b, and for a subsequent call to an input subroutine to wait for input. This can happen if you press CONTROL-P after TTY$IN is called, which causes a quit to PRIMOS and the flushing of the input buffer. When you press START, the next input request waits for a character.

Because FTN cannot call functions without arguments, this routine cannot be called directly from FTN.

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.

TTY$RS

Purpose

This routine is called to clear the user's input and output buffers. A key is passed that contains two bits specifying whether the input and output buffers are to be cleared. This routine takes no action for noninteractive users (such as phantoms and batch jobs).

Usage

```
DCL TTY$RS ENTRY (FIXED BIN, FIXED BIN);

CALL TTY$RS (key, code);
```

Parameters

key

INPUT. The keys indicating whether or not to clear the I/O buffers. Possible key values are:

K$OUTB	Clear output buffer
K$INB	Clear input buffer

code

OUTPUT. 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.

4
Memory Allocation

This chapter describes procedures that allow you to allocate and free blocks of contiguous memory. This is a useful feature in many applications where either the size or the number of data structures is not known until runtime. With help from these procedures, the system allocates only as much memory as is needed.

The first part of this chapter lists procedures for allocating and freeing various classes of dynamic memory. Refer to the Advanced Programmer's Guide for a discussion of these classes. There are pairs of routines for allocating and freeing. Two allocation routines are provided for user-class memory; one indicates errors by returning an error code, the other by raising a condition. Which routine you use depends on the convenience you want. There are also two freeing routines, with the same distinction in error indications.

The second section of this chapter contains specific functions related to the use of command function programs built with BIND (EPFs).

The third section of this chapter lists procedures that tell you how much memory is available.

Most of the routines have an argument of type "pointer". This makes them difficult to use from FORTRAN and COBOL. These languages have no support for pointer-based structures. Also, many routines return a short (2-halfword) pointer. Pascal programs expect a 3-halfword pointer, which is returned differently. Therefore, Pascal programs will not work correctly with these routines.

GENERAL-PURPOSE ALLOCATE AND FREE ROUTINES

This section describes the following subroutines:

Routine	Function
ALOC$S	Allocates memory on the current stack.
STR$AL	Allocates user-class dynamic memory.
STR$AP	Allocates process-class dynamic memory.
STR$AS	Allocates subsystem-class dynamic memory.
STR$AU	Allocates user-class dynamic memory.
STR$FP	Frees process-class dynamic memory.
STR$FR	Frees user-class dynamic memory.
STR$FS	Frees subsystem-class dynamic memory.
STR$FU	Frees user-class dynamic memory.

ALOC$S

Purpose

This routine allocates an area of memory on the current procedure's stack.

Usage

```
DCL ALOC$S (FIXED BIN, POINTER, BIT(1)) OPTIONS (SHORTCALL(4));

CALL ALOC$S (block_size, block_ptr, contig_flag);
```

Parameters

block_size

INPUT. Number of halfwords to allocate.

block_ptr

OUTPUT. Points to allocated storage. If block_size is zero or negative, block_ptr returns the null pointer.

contig_flag

OUTPUT. If true ('1'b), the space was allocated in an area contiguous with the current stack. If false ('0'b), a new segment was allocated for the stack extension.

Discussion

The memory allocated by ALOC$S is found by extending the calling procedure's stack frame. For this reason, the memory remains usable only until the calling procedure returns to its own caller, at which time the memory is automatically de-allocated. The address of the allocated memory should never be passed out to a calling procedure.

ALOC$S must be declared with the attribute OPTIONS (SHORTCALL(4)). This makes the procedure callable only from PL/I. It could be called from PMA, but PMA programmers will find it more convenient to use the single instruction STEX to produce the same result as ALOC$S. SHORTCALL causes the instruction JSXB to be used instead of the PCL instruction. JSXB does not generate a new stack, but operates using space in the caller's stack. This means the procedure can only be called from a module compiled in V-mode.

Loading and Linking Information

V-mode: No special action.

V-mode with unshared libraries: Load NPFTNLB.

R-mode: Not available.

STR$AL

Purpose

This routine allocates space from dynamic memory for user-class storage. It returns an informative error code if a problem occurs, instead of raising a condition (as in STR$AU).

Usage

```
DCL STR$AL ENTRY (FIXED BIN(15), FIXED BIN(31), FIXED BIN(15),
                  FIXED BIN(15)) RETURNS(POINTER) OPTIONS(SHORT);

block_ptr =  STR$AL (reserved, block_size, reserved, code);
```

Parameters

reserved

INPUT. This field must have a value of zero (0).

block_size

INPUT. The size of the block to allocate, in halfwords.

reserved

INPUT. This field must have a value of zero (0).

code

OUTPUT. Standard error code. Possible error codes are:

E$ALSZ Invalid block_size

E$ROOM Insufficient space

E$HPER Corrupt heap

block_ptr

RETURNED VALUE. The pointer to the allocated space.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: Not available.

STR$AP

Purpose

This routine allocates space from process-class storage. If any errors are detected, an appropriate error message is displayed and a condition is signalled.

Usage

```
DCL STR$AP ENTRY (FIXED BIN(31)) RETURNS(POINTER) OPTIONS(SHORT);

block_ptr = STR$AP (block_size);
```

Parameters

block_size

INPUT. The size of the block to allocate, in halfwords.

block_ptr

RETURNED VALUE. Pointer to the allocated space.

Discussion

If any errors are detected, STR$AP signals the condition SYSTEM_STORAGE$. The default action taken by the system is then to re-initialize 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.

STR$AS

Purpose

This routine allocates space from dynamic memory for subsystem-class storage. If any errors are detected, an appropriate error code is returned.

Note

Use STR$AS to allocate dynamic memory space for Prime-supplied subsystems ONLY.

Usage

```
DCL STR$AS ENTRY (FIXED BIN(31), FIXED BIN(15))
                 RETURNS(POINTER) OPTIONS(SHORT);

block_ptr = STR$AS (block_size, code);
```

Parameters

block_size

INPUT. The size (in halfwords) of the block to allocate.

code

OUTPUT. Standard error code. Possible error codes are:

E$BPAR Invalid value for block_size

E$ROOM Insufficient space

E$NSUC Corrupt heap

block_ptr

RETURNED VALUE. Pointer to the allocated space.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: Not available.

STR$AU

Purpose

This routine allocates space from dynamic memory for user-class storage. If an error occurs, a condition is raised.

Usage

```
DCL STR$AU ENTRY (FIXED BIN(31)) RETURNS(POINTER) OPTIONS(SHORT);

block_ptr = STR$AU (block_size);
```

Parameters

block_size

INPUT. Size of the block to allocate (in halfwords).

block_ptr

RETURNED VALUE. Pointer to the allocated space.

Discussion

When a bad block_size is given, this routine raises the ERROR condition. When not enough space can be found in the heap, the routine raises the STORAGE condition. When the heap is found to be corrupted, it raises the HEAP_ERROR$ condition.

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.

STR$FP

Purpose

This routine returns space to process-class storage. If any errors are detected, an appropriate error message is displayed and a condition is signalled.

Usage

```
DCL STR$FP ENTRY (POINTER) OPTIONS(SHORT);

CALL STR$FP (block_ptr);
```

Parameters

block_ptr

INPUT. Pointer to the allocated space.

Discussion

If any errors are detected, STR$FP signals the condition SYSTEM_STORAGE$. The default action taken by the system is then to re-initialize 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.

STR$FR

Purpose

This routine returns space to user-class storage. If any errors are detected, an error code is returned (instead of an error condition as with STR$FU).

Usage

```
DCL STR$FR ENTRY (FIXED BIN(15), POINTER, FIXED BIN(15));

CALL STR$FR (reserved, block_ptr, code);
```

Parameters

reserved

INPUT. Reserved.

block_ptr

INPUT. Pointer to the storage space to be freed.

code

OUTPUT. Standard error code. Possible error codes are:

E$FRER Invalid free request

E$HPER Corrupted heap

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.

STR$FS

Purpose

This routine returns space to subsystem-class storage. If any errors are detected, an appropriate error code is returned.

Usage

```
DCL STR$FS ENTRY (POINTER, FIXED BIN(15));

CALL STR$FS (block_ptr, code);
```

Parameters

block_ptr

INPUT. Pointer to the allocated space.

code

OUTPUT. Standard error code. Possible error codes are:

E$FRER Invalid free request

E$NSUC Corrupted heap

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.

STR$FU

Purpose

This routine returns space to user-class storage. If an error occurs, a condition is raised.

Usage

```
DCL STR$FU ENTRY (POINTER);

CALL STR$FU (block_ptr);
```

Parameters

block_ptr

INPUT. Pointer to block of data to free.

Discussion

When a bad block_ptr is passed, it raises the ERROR condition. When the heap is found to be corrupted, it raises the HEAP_ERROR$ condition.

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.

COMMAND FUNCTION RETURNED DATA ROUTINES

This section describes the following subroutines:

Routine	Function
ALC$RA	Allocates space for EPF function return information.
ALS$RA	Allocates space and sets value of EPF function return information.
FRE$RA	De-allocates space for RPF function return information.

ALC$RA

Purpose

This routine allocates space for EPF (Executable Program Format) function return information. Refer to the Advanced Programmer's Guide for a further discussion of ALC$RA and ALS$RA.

Note

This interface requires the caller to perform pointer-based operations. Fortran or COBOL programs should use the ALS$RA subroutine.

Usage

```
DCL ALC$RA ENTRY (FIXED BIN(31), POINTER);

CALL ALC$RA (space_needed, rtn_fcn_ptr);
```

Parameters

space_needed

INPUT. The total amount of space needed for the return structure (in 16-bit halfwords). It is the sum of the space needed for the return value and the structure version number. See below for the layout of the return structure.

rtn_fcn_ptr

OUTPUT. The pointer to the information to be returned by the function.

Discussion

When a function returns information, it passes the data to the caller via an assignment statement. For an EPF (Executable Program Format) to do this, it must create an indirect pointer and a storage area, so that when the data is returned at execution time it can be stored and accessed by the caller of the function. In order to pass such information to the operating system, an interface (given in the discussion below) defines rtn_fcn_ptr and rtn_fcn_struc.

ALC$RA provides you the space for rtn_fcn_struc; it also returns the value for rtn_fcn_ptr, which you can then pass back to the caller of the EPF function.

Refer to the Advanced Programmer's Guide for a detailed discussion of the following interface.

When using dynamic storage allocation, an EPF program acting as a function (that is, passing back some result to the operating system) must first have the following interface defined:

```
dcl your_epf entry(char(1024) var, fixed bin(15),
                   1, 2 char(32) var,
                      2 fixed bin(15),
                      2 ptr,
                      2, 3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 fixed bin(31),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(11),
                         3 bit(1),
                         3 bit(1),
                         3 bit(14),
                         3 fixed bin(15),
                         3 fixed bin(15),
                         3 bit(1),
                         3 bit(1),
                         3 bit(1),
                         3 bit(13),
                   1, 2 bit(1),
                      2 bit(15),
                   ptr);

call your_epf(command_args, command_status, command_state,
              command_fcn_flags, rtn_fcn_ptr);
```

These arguments are defined as follows:

command_args The entire command line as entered by the user.

command_status The command status returned by the program to the operating system:

= 0 No error
> 0 Fatal error
< 0 Soft error or warning

command_state Information relative to this invocation. It contains, in the order specified:

command name -- Command entered by user.

version -- Current version of the structure of the command state (1 at Rev. 20.2).

vcb_ptr -- Pointer to CPL local variables.

preprocessing_info -- Information relating to what has been preprocessed:

mod_after_date -- If nonzero, then the command processor has found something modified after the given date.

mod_before_date -- If nonzero, then the command processor has found something modified before the given date.

bk_after_date -- If nonzero, then the command processor has found something backed up after the given date.

bk_before_date -- If nonzero, then the command processor has found something backed up before the given date.

type_dir -- If nonzero, a directory has been found that matches a wildcard.

type_segdir -- If nonzero, a segment directory has been found that matches a wildcard.

type_file -- If nonzero, a file has been found that matches a wildcard.

type_acat -- If nonzero, an access category has been found that matches a wildcard.

type_rbf -- If nonzero, a recovery-based file has been found that matches a wildcard.

res1 -- 11 bits with undefined values.

verify_sw -- If nonzero, the -VERIFY option has been given.

botup_sw -- A full treewalk was performed before executing program.

res2 -- 14 bits with undefined values.

walk_from -- Tree level at which the present treewalk started.

walk_to -- Present treewalk level.

in_iteration -- If nonzero, the command processor is currently in an iteration sequence.

in_wildcard -- If nonzero, the command processor is currently in a wildcard sequence.

in_treewalk -- If nonzero, the command processor is currently in a treewalk sequence.

res3 -- 13 bits with undefined values.

command_fcn_flags Information relative to this command function invocation. Its contents in the order specified are:

command_fcn_call -- If nonzero, this program has been called as a command function.

reserved -- 15 bits with undefined values.

rtn_fcn_ptr Pointer to a structure that describes the values returned to the caller of the EPF function. This structure is itself defined as:

```
dcl 1 rtn_fcn_struc,
      2 version fixed bin(15),
      2 value_str char(*) var;
```

Where:

version -- Structure's version (see following discussion).

value_str -- String of 1 to 32767 characters holding the value to be returned.

First obtain the value of rtn_fcn_ptr by calling ALC$RA (or ALS$RA). After the call to ALC$RA, your program must set the version number of rtn_fcn_struc to 0 and copy the value of that structure into value_str. Then the interface sets rtn_fcn_ptr in its main entrypoint's calling sequence and returns to the calling program.

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.

ALS$RA

Purpose

This routine is used both to allocate space from process-class storage for EPF (executable program format) function return information and to set the value of the information. It also assigns the value 0 to the version number within the return function structure. See rtn_function_addr below.

Usage

```
DCL ALS$RA ENTRY (CHAR(*), FIXED BIN(31), POINTER);

CALL ALS$RA (function_result_str, char_size_of_str,
             rtn_function_addr);
```

Parameters

function_result_str

INPUT. The character string that is the result of the program invoked as a function. The string can contain up to 8192 characters.

char_size_of_str

INPUT. The number of characters in function_result_str.

rtn_function_addr

OUTPUT. The address allocated to rtn_fcn_struc. The structure itself has this format:

```
1 rtn_fcn_struc,
        2 version fixed bin(15),
        2 value_str char(*) var;
```

Discussion

The address is returned as a pointer to the EPF function that called ALS$RA; the calling function then stores it for future use.

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.

FRE$RA

Purpose

This routine de-allocates the space designated for the information from the EPF (executable program format) functions. After processing the information returned from functions, the invoker should call this routine to free up space and maintain an efficient command environment.

Usage

```
DCL FRE$RA ENTRY (POINTER);

CALL FRE$RA (rtn_function_ptr);
```

Parameters

rtn_function_ptr

INPUT. Pointer to the space set aside for EPF functions, earlier allocated by ALC$RA or ALS$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.

INFORMATIONAL ROUTINES

This section describes the following subroutines:

Routine	Function
DY$SGS	Returns maximum number of dynamic segments.
ST$SGS	Returns maximum number of static segments.
TL$SGS	Returns highest segment number.

DY$SGS

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the user's profile. This routine retrieves the maximum number of private, dynamic segments allocated to the user.

Usage

```
DCL DY$SGS ENTRY () RETURNS (FIXED BIN(15));

maximum_private_dynamic_segs = DY$SGS ();
```

Parameters

maximum_private_dynamic_segs

RETURNED VALUE. The maximum number of private dynamic segments allocated to the user.

Discussion

This function cannot be called from FTN because it has no parameters.

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.

ST$SGS

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the user's profile. This routine retrieves the maximum number of private, static segments allocated to the user.

Usage

```
DCL ST$SGS ENTRY () RETURNS (FIXED BIN(15));

maximum_private_static_segs = ST$SGS ();
```

Parameters

maximum_private_static_segs

RETURNED VALUE. Maximum number of private static segments allocated to the user.

Discussion

This function cannot be called from FTN because it has no parameters.

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.

TL$SGS

Purpose

This routine is one of several that retrieve EPF-related information from the in-memory copy of the user's profile. This routine retrieves the number of the highest segment that can be allocated to the user.

Usage

```
DCL TL$SGS ENTRY () RETURNS (FIXED BIN);

maximum_private_seg = TL$SGS ();
```

Parameters

maximum_private_seg

RETURNED VALUE. Segment number of the highest private segment that can be allocated to the user.

Discussion

Private segments are allocated from the range 4000-5777 octal (2048-3071 decimal). Therefore, to determine how many segments can be allocated in this range, subtract 2047 from maximum_private_seg.

This function cannot be called from FTN because it has no parameters.

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.

5
Program Control

The first part of this chapter contains routines of general use in controlling the user's command environment and terminating programs.

The second part of this chapter contains routines used for controlling static-mode programs.

The third part of this chapter contains routines used for controlling phantom processes. A phantom is a process that can operate separately from its creator process, and can continue working after the creator has logged out. Phantoms are discussed in detail in the Prime User's Guide.

Several of the routines described here operate by raising (signalling) conditions. The information about these conditions is of use to designers of complex subsystems that communicate between programs. The condition mechanism is described in Chapter 7.

RECURSIVE COMMAND ENVIRONMENT

The recursive command environment provides a fully recursive command processing loop that is also highly modular. The implementation of this environment divides system and user software into two categories: static mode and recursive mode.

Static-mode software

- Allocates its own segments
- Manages its own stack
- Manages its own shared libraries' initialization
- Uses a "memory image" approach; the program is reloaded each time it is called and thus programs cannot be recursively invoked from command level

Recursive-mode software

- Uses the system stack
- Terminates by returning to the calling procedure
- Does not attempt to initialize shared libraries
- Is not reloaded as a memory image each time it is called

User on-units, any procedures they call, and all internal commands are recursive-mode software.

A recursive-mode procedure should terminate by returning, not by calling EXIT. Arguments for recursive-mode commands are passed as parameters and are not obtained from a static buffer. Error information is passed by setting a return parameter (error code), printing an error message and returning, or by signalling a condition. The ERRRTN call must not be used. ERRPR$ can be used with any of its three valid keys; see the discussion with the routine description in Chapter 3. Recursive-mode programs and library routines are implemented as Executable Program Format (EPF) files. Executable Program Format is discussed in detail in Volume II of the Subroutines Reference Guide.

PHANTOM PROCESSES AND LOGOUT NOTIFICATION

A phantom is a process that can operate separately from its creator process, and can continue working after the user has logged out. Phantoms are discussed in detail in the Prime User's Guide.

Logout Notification for Phantoms

Logout notification provides the creator of a phantom process with information about the phantom's activities. This information is compiled at phantom logout time and sent to the creator. This is known as notification.

Normally, the information will be displayed at the creator's terminal. The information contains the phantom's user number, the time of day of logout, the elapsed time, CPU time, I/O time spent by the phantom, and an error code indicating normal or abnormal logout. Normal logout occurs when a phantom completes with a LOGOUT command. All other logouts will generate abnormal logout.

Logout information will be compiled at this time and sent to the creator. If a user is logged in as more than one process, the information will only be sent to the process from which the phantom was created. If the creator of the phantom has logged out while the phantom was running, the information will not be sent. This means that once a user has logged out, the phantom will not notify the user of logout even if the user logs back in.

Sometimes it becomes necessary for a user to record the phantom logout information via a program. The logout notification system provides two subroutines that allow for this event. The first subroutine, LON$CN, allows a user to turn logout notification off or on. The second subroutine, LON$R, allows a user to fetch phantom logout information instead of having the information written to a terminal.

When LON$CN is called to turn off logout notification, phantom logout information is automatically queued for future access. This allows users to turn off logout notification without having to worry about either the condition of their terminal screen or the loss of the status of their phantoms.

When LON$CN is requested to turn on logout notification, any pending logout information is written to the user's terminal.

As mentioned above, a user may fetch phantom logout information by invoking LON$R. Normally, logout notification is enabled, and invoking LON$R will have no effect. Therefore, when using LON$R, logout notification should be turned off by invoking LON$CN.

When logout notification occurs, a system default condition handler or on-unit named PH_LOGO$ is invoked to write the information upon the creator's terminal. This on-unit is also invoked when LON$CN is requested to turn on logout notification. Users who do not ever wish to see logout information written upon their terminal should create their own on-unit for PH_LOGO$. This user-defined PH_LOGO$ will usually call LON$R to fetch phantom logout information.

COMMAND LEVEL CONTROL ROUTINES

This section describes the following subroutines:

Routine	Function
CMLV$E	Calls a new command level after an error.
COMLV$	Calls a new command level.
EXIT	Returns to PRIMOS.
ICE$	Initializes the command environment.
SETRC$	Records command error status.
SS$ERR	Signals an error in a subsystem.

CMLV$E

Purpose

This routine causes a new command level to be called after an error occurs.

Usage

```
DCL CMLV$E ENTRY;

CALL CMLV$E;
```

Parameters

There are no parameters.

Discussion

When CMLV$E is called, a PRIMOS routine called the command listener does the following: it pauses command input, displays the error prompt, waits for input, forces terminal output on, and enables quits. The CMLV$E subroutine returns to the caller only after you issue a START command from the new command level.

Compare this to COMLV$, which should be called to perform similar functions in situations where there has not been an 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: Not available.

COMLV$

Purpose

This routine causes a new command level to be called.

Usage

```
DCL COMLV$ ENTRY;

CALL COMLV$;
```

Parameters

There are no parameters.

Discussion

When COMLV$ is called, a PRIMOS routine called the command listener displays the ready prompt and waits for input. Only after you issue the START command from that command level will the COMLV$ subroutine return to the caller.

Compare CMLV$E, which should be called to perform similar functions in error situations.

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.

EXIT

Purpose

The EXIT subroutine provides a way to return from a user program to the PRIMOS command processor.

Usage

```
DCL EXIT ENTRY;

CALL EXIT;
```

Parameters

There are no parameters.

Discussion

EXIT is intended for use from a static-mode program. EPF (Executable Program Format) programs should terminate by using the RETURN statement in the main program, but may call EXIT if desired. For example, it may be convenient to call EXIT to terminate the program from a subroutine many call levels deep. In EPF programs, CALL EXIT is much less efficient than using a RETURN.

When EXIT causes a return to the command level, the PRIMOS command processor prints the ready prompt (initially OK, or OK:) at the terminal and awaits a user command. If EXIT is called from a static-mode program, the user may open or close files or switch directories, and restart a program at the next statement by typing S (START). If EXIT is called from an EPF, it signals the STOP$ condition and disables continuation using the START command.

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.

ICE$

Purpose

This routine initializes the command environment.

Usage

```
DCL ICE$ ENTRY;

CALL ICE$;
```

Parameters

There are no parameters.

> **Caution**
>
> Avoid using this subroutine! It may affect the integrity of subsystems, including Prime data management products. CLEANUP$ on-units on the stack are not invoked. Consequently, it should be used only when the stack has clearly been damaged.

Discussion

ICE$ closes all open files, including the command output file, and resets your environment to its initial state. The routine never returns, and the invoking program is terminated. If you are working in a subdirectory during an ICE$, you are returned to your origin UFD.

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.

SETRC$

Purpose

This routine records the code you give it. Later, when the program exits, the system regards the code you gave as the error status. This routine does not cause an immediate return to the calling software.

Usage

```
DCL SETRC$ ENTRY (FIXED BIN [, BIT(1)ALIGNED] );

CALL SETRC$ (severity_code [, abort_flag] );
```

Parameters

severity_code

INPUT. The severity code to return to the invoker of this program.

abort_flag

OPTIONAL INPUT. Value is '1'b if the command file (if any) is to be aborted, and '0'b if it is not to be aborted. (This flag will make no difference if this command was invoked by a CPL procedure.)

Discussion

If severity_code is less than or equal to 0, then abort_flag is ignored, and the command file is never aborted.

If severity_code is greater than 0, and abort_flag is omitted or '0'b, the condition SETRC$ is signalled. The default on-unit for SETRC$ records the occurrence of the event and returns.

SETRC$ is intended for use from static-mode programs only. EPF (Executable Program Format) programs set the status code by using an output parameter.

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.

SS$ERR

Purpose

This routine signals an error in a subsystem. It is intended to terminate the program immediately if it is being used in a phantom.

Usage

```
DCL SS$ERR ENTRY;

CALL SS$ERR;
```

Parameters

There are no parameters.

Discussion

If a command input file is active, the condition SUBSYS_ERR$ is raised. Raising this condition usually results in the termination of the caller by means of a nonlocal GOTO back to the command processor. If you are interactive, SS$ERR simply returns.

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.

STATIC-MODE SAVE AND RESTORE ROUTINES

This section describes the following subroutines:

Routine	Function
REST$$	Restores an R-mode executable image.
RESU$$	Restores and resumes an R-mode executable image.
SAVE$$	Saves an R-mode executable image.

REST$$

Purpose

REST$$ reads R-mode executable code from a file in the current UFD into memory.

Usage

```
DCL REST$$ ENTRY ((9)FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN);

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

Parameters

vector

OUTPUT. A nine-halfword array set by REST$$. vector(1) is set to the first location in memory to be restored. vector(2) is set to the last location to be restored. The array is set as follows:

vector(1)	Set to first location in memory to be restored
vector(2)	Set to last location in memory to be restored
vector(3)	Saved P register
vector(4)	Saved A register
vector(5)	Saved B register
vector(6)	Saved X register
vector(7)	Saved keys
vector(8)	Not used
vector(9)	Not used

filnam

INPUT. The name of the file containing the executable image.

namlen

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

code

OUTPUT. Standard error code.

Note

Use the PRIMOS command SEG to restore segmented V-mode runfiles from a segment directory. Use the PRIMOS command RESUME, or the EPF (executable program format) handling routines described in Volume II of the Subroutines Reference Guide, to restore a runfile from an EPF file.

Discussion

The saved parameters for a file previously written to the disk by the SAVE$$ routine, the SAVE command, or the SAVE subcommand of the R-mode loader, are loaded into the nine-halfword array vector. The code itself is then loaded into memory using the starting and ending addresses provided by vector(1) and vector(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.

RESU$$

Purpose

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

Usage

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

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

Parameters

filnam

INPUT. The name of the file containing the code.

namlen

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

Discussion

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

Note

Use the PRIMOS command SEG to restore segmented V-mode runfiles from a segment directory. Use the PRIMOS command RESUME, or the EPF (executable program format) handling routines described in Volume II of the SUBROUTINES REFERENCE GUIDE, to restore a runfile from an EPF file.

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.

SAVE$$

Purpose

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

Usage

```
DCL SAVE$$ ENTRY ((9)FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN);

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

Parameters

vector

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

vector(1)	Set to an integer that is the first location in memory to be saved
vector(2)	Set to last location to be saved
vector(3)	Saved P register
vector(4)	Saved A register
vector(5)	Saved B register
vector(6)	Saved X register
vector(7)	Saved keys
vector(8)	Not used
vector(9)	Not used

filnam

INPUT. The name of the file to contain the code.

namlen

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

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: No special action.

PHANTOM PROCESS CONTROL ROUTINES

This section describes the following subroutines:

Routine	Function
LON$CN	Switches logout notification on or off.
LON$R	Reads logout notification information.
PHNTM$	Starts a phantom process.

LON$CN

Purpose

This subroutine is used to turn off, or turn on, logout notification. When notification is turned off, phantom logout information is queued (first-in/first-out). When notification is turned on, queuing is not performed, but if there is any logout notification data to be received, the default condition, PH_LOGO$, is raised.

See the discussion of LON$R for more information.

Usage

```
DCL LON$CN ENTRY (FIXED BIN);

CALL LON$CN (key);
```

Parameters

key

INPUT. Software interrupt status key:

0	Notify off
1	Notify on

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.

LON$R

Purpose

This subroutine fetches or transfers logout information from storage to a designated target area; it will do this unless it finds no information to transfer.

Usage

```
DCL LON$R ENTRY (POINTER, FIXED BIN, BIT, FIXED BIN);

CALL LON$R (msgptr, msglen, more, code);
```

Parameters

msgptr

INPUT -> OUTPUT. Area of memory in which to store the message. Its format is shown in the Discussion section.

msglen

INPUT. Length of area in which to store message.

more

OUTPUT. Standard code.

0	No messages left on queue
1	Messages left on queue

code

OUTPUT. Standard error code.

E$NDAT	No data found in queue
E$BFTS	Some information lost during transfer (msglen less than actual message length)

Discussion

The target area is designated by the argument msgptr. The size of the area pointed to by msgptr is designated by the argument msglen. The area should be at least six halfwords in length. If it is shorter than this, LON$R will only fetch as much information as msglen can hold.

The format of the target area is as follows:

Halfword Number	Information
1	Phantom's user number (fixed bin(15))
2	Time of logout (fixed bin(15))
3	Connect (elapsed) time in minutes (fixed bin(15))
4	CPU time in seconds (fixed bin(15))
5	I/O time in seconds (fixed bin(15))
6	Logout condition code (fixed bin(15)):
	0 Normal logout
	1 Abnormal logout

LON$R also passes back to its caller an indication whether there have been more phantom logouts with their information stored in a queue. This indication is contained within the argument more.

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.

PHNTM$

Purpose

This subroutine allows a process to start a phantom using either a command input file or a CPL file. Use the .CPL suffix for CPL files only; non-CPL programs must not have a .CPL suffix.

Usage

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

CALL PHNTM$ (cplflg, filename, name_len, funit, phant_user,
                  code, args, args_len);
```

Parameters

cplflg

INPUT. Source of the process: if 1, then a CPL program is being started as a phantom; if 0, then a command input file is being started as a phantom.

filename

INPUT. The name of the file to be started as a phantom. The filename must end in .CPL if the program is a CPL program. Use the .CPL suffix for CPL programs only.

name_len

INPUT. The number of characters in filename.

funit

INPUT. The file unit on which to open the phantom file.

phant_user

OUTPUT. The user number of the phantom.

code

OUTPUT. Standard error code; 0 means no error.

args

INPUT. The arguments for a CPL phantom; a dummy argument must be given for non-CPL phantoms.

args_len

INPUT. The number of characters in args; a dummy argument must be given for non-CPL phantoms.

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
Conversion Routines and Other Utilities

The first two sections of this chapter contain subroutines that convert data from one form to another. The section NUMERIC CONVERSION ROUTINES describes routines that convert character strings into numbers. The section DATE CONVERSION ROUTINES describes routines that convert date-time information from one format to another.

The third section, OTHER ROUTINES, describes routines that manipulate data in ways not covered by other chapters of this volume. They perform a binary search, encrypt a password, store and retrieve characters in arrays, parse a character string into tokens, transfer output to a buffer, move a block of memory, produce unique strings for identification purposes, or match a name against a wildcard specification.

NUMERIC CONVERSION ROUTINES

This section describes the following subroutines:

Routine	Function
CH$FX1	Converts string (decimal) to 16-bit integer.
CH$FX2	Converts string (decimal) to 32-bit integer.
CH$HX2	Converts string (hexadecimal) to 32-bit integer.
CH$OC2	Converts string (octal) to 32-bit integer.

CH$FX1

Purpose

CH$FX1 converts a character string of any length into a FIXED BIN (15) number. The string is interpreted as a decimal number.

Usage

```
DCL CH$FX1 ENTRY (CHAR (*) VAR, FIXED BIN (15) [, FIXED BIN (15)]);

CALL CH$FX1 (string_to_convert, result [, nonstandard_code]);
```

Parameters

string_to_convert

INPUT. CHARACTER (*) VARYING string that is to be converted. Leading and trailing blanks are permitted. The minus sign (-) is permitted, but the plus sign (+) is not. The string must represent an integer; the decimal point is an invalid character. If the numeric value of the string is greater than 32767 or less than -32767, the result is undefined.

result

OUTPUT. FIXED BINARY (15) number produced by the conversion. Zero if the string was null or illegal.

nonstandard_code

OPTIONAL OUTPUT. Nonstandard error code. If this parameter is not supplied and an error occurs, the CONVERSION condition is signalled. The possible values of the code are

1 String contains embedded blanks
2 Overflow
3 Bad character in conversion
4 Illegal field

Discussion

CH$FX1 is part of the PRIMOS binary conversion package. Other modules in this package include

- CH$FX2, like CH$FX1 except that it returns a FIXED BIN (31) value
- CH$OC2, like CH$FX2 except that it treats the string as octal
- CH$HX2, like CH$FX2 except that it treats the string as hexadecimal

All have the same basic calling sequence.

These routines are useful if you have a file that contains numbers stored as character strings and you wish to perform computations on the numbers. If you use the error code argument, you have more control over input errors than you do with the formatted I/O statements in most languages. And although PL/I automatically performs a type conversion if you assign a character string to a numeric variable, it also signals the CONVERSION condition for bad input format. These subroutines, however, enable you to gain information about input errors while you avoid incurring a runtime 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: Not available.

CH$FX2

Purpose

CH$FX2 converts a character string of any length into a FIXED BIN (31) number. The string is interpreted as a decimal number.

Usage

```
DCL CH$FX2 ENTRY (CHAR (*) VAR, FIXED BIN (31) [, FIXED BIN (15)]);

CALL CH$FX2 (string_to_convert, result [, nonstandard_code]);
```

Parameters

string_to_convert

INPUT. CHARACTER (*) VARYING string that is to be converted. Leading and trailing blanks are permitted. The minus sign (-) is permitted, but the plus sign (+) is not. The string must represent an integer; the decimal point is an invalid character. If the numeric value of the string is greater than 2147483647 or less than -2147483647, the result is undefined.

result

OUTPUT. FIXED BINARY (31) number produced by the conversion. Zero if the string was null or illegal.

nonstandard_code

OPTIONAL OUTPUT. Nonstandard error code. If this parameter is not supplied and an error occurs, the CONVERSION condition is signalled. The possible values of the code are

1 String contains embedded blanks
2 Overflow
3 Bad character in conversion
4 Illegal field

Discussion

CH$FX2 is part of the PRIMOS binary conversion package. Other modules in this package include CH$FX1, CH$HX2, and CH$OC2. See CH$FX1 for a description of their functions.

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.

CH$HX2

Purpose

CH$HX2 converts a character string of any length into a FIXED BIN (31) number. The string is interpreted as a hexadecimal number.

Usage

```
DCL CH$HX2 ENTRY (CHAR (*) VAR, FIXED BIN (31) [, FIXED BIN (15)]);

CALL CH$HX2 (string_to_convert, result [, nonstandard_code]);
```

Parameters

string_to_convert

INPUT. CHARACTER (*) VARYING string that is to be converted. Leading and trailing blanks are permitted. The minus sign (-) is permitted, but the plus sign (+) is not. The string must represent an integer; the decimal point is an invalid character. If the numeric value of the string is greater than 7FFFFFF or less than -7FFFFFF, the result is undefined.

result

OUTPUT. FIXED BINARY (31) number produced by the conversion. Zero if the string was null or illegal.

nonstandard_code

OPTIONAL OUTPUT. Nonstandard error code. If this parameter is not supplied and an error occurs, the CONVERSION condition is signalled. The possible values of the code are

1 String contains embedded blanks
3 Bad character in conversion

Discussion

CH$HX2 is part of the PRIMOS binary conversion package. Other modules in this package include CH$FX1, CH$FX2, and CH$OC2. See CH$FX1 for a description of their functions.

CH$HX2 interprets the input string as the representation of a hexadecimal number. It converts the string to a FIXED BIN (31) number,

which can then be printed out as a decimal, octal, or hexadecimal number, depending on the output procedure you use. The input string FFF would print in decimal form as 4095. All ten digits, as well as the uppercase characters A through F, are valid. Lowercase letters are illegal and receive error code 3.

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.

CH$OC2

Purpose

CH$OC2 converts a character string of any length into a FIXED BIN (31) number. The string is interpreted as an octal number.

Usage

```
DCL CH$OC2 ENTRY (CHAR (*) VAR, FIXED BIN (31) [, FIXED BIN (15)]);

CALL CH$OC2 (string_to_convert, result [, nonstandard_code]);
```

Parameters

string_to_convert

INPUT. CHARACTER (*) VARYING string that is to be converted. Leading and trailing blanks are permitted. The minus sign (-) is permitted, but the plus sign (+) is not. The string must represent an integer; the decimal point is an invalid character. If the numeric value of the string is greater than 17777777777 or less than -17777777777, the result is undefined.

result

OUTPUT. FIXED BINARY (31) number produced by the conversion. Zero if the string was null or illegal.

nonstandard_code

OPTIONAL OUTPUT. Nonstandard error code. If this parameter is not supplied and an error occurs, the CONVERSION condition is signalled. The possible values of the code are

1 String contains embedded blanks
3 Bad character in conversion

Discussion

CH$OC2 is part of the PRIMOS binary conversion package. Other modules in this package include CH$FX1, CH$FX2, and CH$HX2. See CH$FX1 for a description of their functions.

CH$OC2 interprets the input string as the representation of an octal number. It converts the string to a FIXED BIN (31) number, which can

then be printed out as a decimal, octal, or hexadecimal number, depending on the output procedure you use. The input string 777 would print in decimal form as 511. The digits 8 and 9 are illegal and receive error code 3.

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.

DATE CONVERSION ROUTINES

This section describes the following subroutines:

Routine	Function
CV$DQS	Converts binary date to quadseconds.
CV$DTB	Converts ASCII date to binary format.
CV$FDA	Converts binary date to ISO format.
CV$FDV	Converts binary date to "visual" format.
CV$QSD	Converts quadsecond date to binary format.

CV$DQS

Purpose

CV$DQS converts a coded binary date string to quadseconds. One quadsecond equals 4 seconds.

Usage

```
DCL CV$DQS ENTRY (FIXED BIN(31), FIXED BIN(31));

CALL CV$DQS (fs_date, quadseconds);
```

Parameters

fs_date

INPUT. The date to be converted, in FS (File System) format. The format of a 32-bit encoded FS-format date is described in Appendix C. You obtain this formatted date by calling the DATE$ system-information subroutine.

quadseconds

OUTPUT. Date as expressed in quadseconds since January 1, 1901 midnight.

Discussion

CV$DQS is part of the PRIMOS standard date package. It takes a standard FS-format bit-encoded date and converts it to absolute quadseconds since January 1, 1901 midnight (01-01-01.00:00:00).

You can use CV$DQS to get dates into numeric form so that you can perform computations on them. For simple comparisons of dates, you can use the FS-format date returned by DATE$.

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.

CV$DTB

Purpose

CV$DTB converts an ASCII-format date to binary format.

Usage

```
DCL CV$DTB ENTRY (CHAR(128) VAR, FIXED BIN(31), FIXED BIN);

CALL CV$DTB (ascii_date, fs_date, code);
```

Parameters

ascii_date

INPUT. The ASCII-format date to be converted. Legal formats are described below.

fs_date

OUTPUT. The bit-encoded FS-format date returned. The format of a 32-bit encoded FS-format date is described in Appendix C.

code

OUTPUT. Standard error code. (See Chapter 1 for information about the standard error codes.) The possible value is

E$BPAR The passed date string is illegal

Discussion

CV$DTB is part of the PRIMOS standard date package. It converts an ASCII-format date to FS (bit-encoded) format. Standard ASCII-format dates can have any of the following three formats:

```
YY-MM-DD.HH:MM:SS{.DOW}           (ISO format)
MM/DD/YY.HH:MM:SS{.DOW}           (USA format)
DD MMM YY HH:MM:SS{Day-of-week}   (Visual format)
```

Omitted date fields are replaced by today's date information; omitted time fields are replaced by zeros. If the string is null, zero is returned. The day-of-week field is checked for consistency only.

CV$DTB is useful if you need to compare dates that may be stored in different ASCII formats. Once you convert them to FS format, you can perform comparisons on them.

If you need to obtain the current date and time in FS format, use the DATE$ system-information subroutine.

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.

CV$FDA

Purpose

CV$FDA converts a coded binary date string to ISO format.

Usage

```
DCL CV$FDA ENTRY (FIXED BIN(31), FIXED BIN, CHAR(21));

CALL CV$FDA (fs_date, day_of_week, formatted_date);
```

Parameters

fs_date

INPUT. The date to be converted, in FS (File System) format. The format of a 32-bit encoded FS (File System)-format date is described in Appendix C. You obtain this formatted date by calling the DATE$ system-information subroutine.

day_of_week

OUTPUT. A number corresponding to the day of the week. Sunday is 0, Monday is 1, and so on.

formatted_date

OUTPUT. ASCII-format date in ISO format, as described below.

Discussion

CV$FDA is part of the PRIMOS standard date package. It converts an FS (File System)-format date string to ISO format.

ISO format dates are designed primarily for machine readability. Dates that are to be read primarily by people should be converted with CV$FDV.

The date returned is in the format "YY-MM-DD.HH:MM:SS.DOW". An example is

```
86-04-15.17:05:36.Tue
```

If the passed date is illegal, formatted_date is set to ** invalid date **, and day_of_week is set to -1.

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.

CV$FDV

Purpose

CV$FDV converts a coded binary date string to "visual" format.

Usage

```
DCL CV$FDV ENTRY (FIXED BIN(31), FIXED BIN, CHAR(28) VAR);

CALL CV$FDV (fs_date, day_of_week, formatted_date);
```

Parameters

fs_date

INPUT. The date to be converted, in FS (File System) format. The format of a 32-bit encoded FS-format date is described in Appendix C. You obtain this formatted date by calling the DATE$ system-information subroutine.

day_of_week

OUTPUT. A number corresponding to the day of the week. Sunday is 0, Monday is 1, and so on.

formatted_date

OUTPUT. ASCII-format date in visual format, as described below.

Discussion

CV$FDV is part of the PRIMOS standard date package. It converts an FS-format date string to "visual" format.

Visual format dates are designed primarily to be read by users. Because they contain blanks and are not ordered in a strictly decreasing way, they are not particularly suited for machine readability. Dates that must be machine-readable should be converted with CV$FDA.

The date returned is in the format "DD MMM YY HH:MM:SS day_of_week". An example is

```
15 Apr 86 17:05:36 Tuesday
```

If the passed date is illegal, formatted_date is set to ** invalid date **, and day_of_week is set to -1.

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.

CV$QSD

Purpose

CV$QSD converts a date and time in quadsecond form into FS (File System) format. One quadsecond equals 4 seconds. CV$QSD is the reverse of CV$DQS.

Usage

```
DCL CV$QSD ENTRY (FIXED BIN(31), FIXED BIN(31));

CALL CV$QSD (quadseconds, fs_date);
```

Parameters

quadseconds

INPUT. The date to be converted, expressed in quadseconds since January 1, 1901 midnight. You usually obtain this value by calling the DATE$ function and then converting its output to quadseconds with CV$DQS.

fs_date

OUTPUT. The date in FS (File System) format. The format of a 32-bit encoded FS-format date is described in Appendix C.

Discussion

CV$QSD is part of the PRIMOS standard date package. It takes a date in absolute quadseconds since January 1, 1901 midnight (01-01-01.00:00:00) and converts it to standard FS-format bit-encoded date format.

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.

OTHER ROUTINES

This section describes the following subroutines:

Routine	Function
BIN$SR	Perform binary search in ordered table.
ENCRYPT$	Encrypt login validation passwords.
GCHAR	Get a character from an array.
GT$PAR	Parse character string into tokens.
IOA$RS	Provide free-format output to a buffer.
MOVEW$	Move a block of memory.
NAMEQ$	Compare two character strings.
SCHAR	Store a character into an array location.
UID$BT	Return unique bit string.
UID$CH	Convert UID$BT output into character string.

BIN$SR

Purpose

BIN$SR performs a binary search in an ordered table kept in part of a segment. The table consists of fixed-size entries indexed by a varying character string. If the routine finds the entry searched for, it returns a pointer to the entry. If it does not find it, it indicates where the missing entry should be inserted into the table. There are three restrictions:

1. The table must fit in a 64K halfword (128K byte) segment.
2. All entries must be the same size.
3. All offsets in the segment must be zero modulo the entry size in halfwords.

Usage

```
DCL BIN$SR ENTRY(CHAR(*) VAR, FIXED BIN, PTR, PTR, PTR,
                                                FIXED BIN);

CALL BIN$SR(entry, entry_size, start_ptr, end_ptr, spot_ptr,
                                                code);
```

Parameters

entry

INPUT. A varying character string that contains the index name of the entry to be searched for.

entry_size

INPUT. The size of each entry in halfwords, including the space for the index name.

start_ptr

INPUT. A pointer to the first entry in the table.

end_ptr

INPUT. A pointer to the last entry in the table.

spot_ptr

OUTPUT. A pointer either to the entry or to the place to insert the entry.

code

OUTPUT. A nonstandard error code with the following meanings:

Value	Meaning
0	The entry was found, and spot_ptr points to it.
1	The entry was not found, and spot_ptr points to where it should be inserted.
2	The arguments to the call were bad; either start_ptr and end_ptr did not point to the same segment, or the halfword offset of either pointer was not zero modulo entry_size.
3	The entry was not found, and the place to insert it, pointed to by spot_ptr, is not in the current segment. This means that the segment is full.

Discussion

This routine can also be used to handle a table in which the indices are integers rather than varying character strings. The following data structure should be used:

```
DCL 1 entry,
      2 lenc FIXED BIN,
      2 name FIXED BIN,
      2 info FIXED BIN;
```

In this structure, entry.lenc is the length of the index in bytes. entry.name is the index; it can be either FIXED BIN(15) or FIXED BIN(31). If it is FIXED BIN(15), then entry.lenc is 2; if it is FIXED BIN(31), then entry.lenc is 4. entry.info is arbitrary; it can be any type or size, not just FIXED BIN. The entry length for this structure is (1 + size(name) + size(info)), but only the name field is used in locating the entry.

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.

ENCRYPT$

Purpose

ENCRYPT$ encrypts login validation passwords for use by the User Registration feature of PRIMOS. Users who need a one-way password encryption algorithm may find it useful.

Usage

```
DCL ENCRYPT$ ENTRY (CHAR(16) VAR) RETURNS (CHAR(16));

encrypted_password = ENCRYPT$ (unencrypted_password);
```

Parameters

unencrypted_password

INPUT. An ASCII login validation password up to 16 characters long.

encrypted_password

RETURNED VALUE. The encrypted value of the unencrypted password.

Discussion

Login validation passwords may contain any characters other than PRIMOS reserved characters. (See the Prime User's Guide for a list of these characters.) Lowercase alphabetic characters are mapped to uppercase.

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.

GCHAR

Purpose

GCHAR gets a character from an array. Its counterpart is SCHAR, which stores a character in an array. SCHAR is described later in this section.

Since GCHAR is strictly a FORTRAN tool, its Usage information is given in FORTRAN format.

Usage

```
INTEGER*2 char, array(1), index

char = GCHAR(LOC(array), index)
```

Parameters

LOC(array)

INPUT. A pointer to the array of characters from which the character is to be retrieved.

index

INPUT/OUTPUT. Index of the location of char in the array. Incremented by 1 after each call to GCHAR.

char

RETURNED VALUE. The character returned, in the right-hand byte of a 16-bit integer.

Discussion

GCHAR is helpful in retrieving character information for a FORTRAN program.

You must load the pointer index with position (X - 1) in order to get the character from position X in the array. For example, if the character is in position 1, then you must initialize index to 0. After the operation, index is incremented by 1.

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.

GT$PAR

Purpose

The subroutine GT$PAR is used to parse a character string into tokens separated by three types of characters. The three types are white spaces, quote characters, and break characters. A single token is returned by each call to GT$PAR.

Usage

```
DCL GT$PAR ENTRY(BIT(16) ALIGNED,
                 CHAR(*) VAR,
                 CHAR(*) VAR,
                 CHAR(*) VAR,
                 CHAR(*) VAR,
                 CHAR(*) VAR,
                 FIXED BIN,
                 1,
                   2,
                     3 BIT(11),
                     3 BIT(1),
                     3 BIT(1),
                     3 BIT(1),
                     3 BIT(1),
                     3 BIT(1),
                   2 CHAR(1) ALIGNED,
                 FIXED BIN);

CALL GT$PAR(key, white, quote, break, source_str, token_str,
            token_str_size, info, next_char);
```

Parameters

key

INPUT. A bit string of length 16. Overlaying it is the following structure:

```
1 key,
   2 mbz BIT(11),
   2 leave_trailing_white_space BIT(1),
   2 no_comment BIT(1),
   2 quote_cont BIT(1),
   2 keep_quotes BIT(1),
   2 no_shift BIT(1);
```

key.mbz

Reserved for future expansion.

key.leave_trailing_white_space

'1'B tells GT$PAR not to skip white space at the end of the token. This will cause the value returned in info.delimiter (see below) to be a white space character, even if there is a break character after the white space character(s). Next_char will point to the character after the first white space character found.

key.no_comment

'0'B tells GT$PAR that the character sequence /* is to signal the end of the line and the start of a comment. '1'B means that no comment delimiter checking is done.

key.quote_cont

'1'B tells GT$PAR to assume that the source character string has an info.delimiter before the first character. This is useful in handling a quoted token that spans multiple strings.

key.keep_quotes

'1'B tells GT$PAR not to remove one level of quote characters after processing them. This means that a quoted token can be correctly reprocessed by another parser as a single literal token.

key.no_shift

'1'B tells GT$PAR not to convert nonquoted lowercase characters to uppercase. '0'B tells it to convert nonquoted lowercase characters to uppercase.

white

INPUT. A varying character string containing all the characters that are to be considered as a white space character. There can be any number and mixture of white space characters between tokens. Any leading and/or trailing white space character(s) are removed from a token.

quote

INPUT. A varying character string containing all the characters that are to be considered as quote characters. All characters between a matched pair of quote characters (including different quote characters) are treated literally. If there are two of the current quote characters in a row, then a single quote character

will be placed in the token and will not be considered the end of the quoted string. (If the key.keep_quotes bit is a '1'B, then all quotes will be kept). For example, if the quote characters were ' and ", then each of the following strings would be considered a single token:

String	Token
'foo bars'' inc.'	foo bars' inc.
foo' bars" 'inc.	foo bars" inc.
"It was John's ball ..."	It was John's ball ...
" a ''mix'' of ""quotes"""	a ''mix'' of "quotes"

break

INPUT. A varying character string containing all the characters that are to be considered as a break character. There is at most one break character between each token. Since a single break character always separates tokens, two break characters in a row have a null token between them. Since leading white space characters are ignored (see above), there can be any number of white space characters between two break characters and that token will still be null.

source_str

INPUT. A varying character string containing the text to be parsed.

token_str

OUTPUT. A varying character string into which GT$PAR will place the token.

token_str_size

INPUT. The maximum length of token_str in characters. If the token is longer than this, it will be truncated, and info.flags.truncated (see below) will be set to '1'B.

info

INPUT/OUTPUT. The following structure, into which GT$PAR will place information about the token returned in token_str:

```
1 info,
   2 flags,
      3 mbz BIT(11),
      3 partial BIT(1),
      3 has_quotes BIT(1),
      3 truncated BIT(1),
      3 delimiter_eol BIT(1),
      3 eol BIT(1),
   2 delimiter CHAR(1) ALIGNED;
```

info.flags.mbz

Reserved for future expansion.

info.flags.partial

'1'B if there was no closing quote for the current token. The quote character is placed in info.delimiter.

info.flags.has_quotes

'1'B if the token is a quoted string.

info.flags.truncated

'1'B if and only if the token was too long to fit into token_str.

info.flags.delimiter_eol

'1'B if this token was delimited by the end of the string.

info.flags.eol

'1'B if there is no token available because the end of source_str has been reached.

info.delimiter

If info.flags.partial is '1'B, then info.delimiter is the quote character. If info.flags.delimiter_eol is '1'B, then info.delimiter is undefined. Otherwise, info.delimiter is the character that delimited the end of the token.

Only if key.quote_cont is '1'B is info.delimiter valid as input to GT$PAR; info.flags is never used for input. If key.quote_cont is '1'B, then info.delimiter contains the current quote character for a quoted token that spans multiple strings.

next_char

INPUT/OUTPUT. The index of the next character to be examined in the source string (character 1 is the first character in the string). If you wish to start parsing at the start of the string, next_char should be set to 1 before the call to GT$PAR. After GT$PAR returns, next_char will point to the character after the delimiting character. This means that the current place in the string can be saved and a particular token can be reparsed if so desired.

Discussion

Any of the white, quote, or break arguments can be the null string. The null string means that there are none of that type of delimiter.

Example:

```
DCL TOKEN CHAR(40) VAR;
DCL NEXT FIXED BIN;
NEXT = 1;
CALL GT$PAR('0'B, ' ', '"', '.', ' A line.', TOKEN, 40, INFO,
            NEXT);
```

The first time the CALL statement is executed, it returns NEXT = 4, all info.flags = '0'B, info.delimiter = ' ', and TOKEN = 'A'.

If the CALL statement is executed again, it returns NEXT = 9, all info.flags = '0'B, info.delimiter = '.', and TOKEN = 'LINE'.

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.

IOA$RS

Purpose

IOA$RS provides free-format output to a buffer. It is similar to IOA$, which provides free-format output to the terminal. IOA$ is described in Chapter 3 of this volume.

Usage

```
DCL IOA$RS ENTRY (CHAR(*), FIXED BIN, FIXED BIN, CHAR(*), FIXED BIN
                          [, any type, ... any type]);

CALL IOA$RS (buffer, bufsize, buflen, control, conlen
                          [, arg1, ... argn]);
```

Parameters

buffer

OUTPUT. The character string into which IOA$RS writes the formatted text.

bufsize

INPUT. The capacity of buffer, in characters: that is, buffer must be able to hold a maximum of bufsize characters. buffer is padded with blanks to this stated capacity if the length of the generated text is less than bufsize.

buflen

OUTPUT. The number of characters of text generated by the formatting and conversion operations.

control

INPUT. A character string that specifies both the literal text to be output and the conversion operations to be performed on the arguments. For information on the format of this string, see the discussion of IOA$.

conlen

INPUT. The length of control, in number of characters. For more information, see the discussion of IOA$ in Chapter 3.

arg1, ... argn

OPTIONAL INPUT. Optional arguments, which can be of any data type. For more information, see the discussion of IOA$ in Chapter 3.

Discussion

IOA$RS is identical to IOA$ except that it puts the formatted text into a character buffer variable, rather than writing it directly to the terminal. In addition, the length of the buffer is specified by the calling program, whereas IOA$ imposes a 400-character limit on output volume.

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.

MOVEW$

Purpose

MOVEW$ moves a block of memory efficiently from one place to another.

Usage

```
DCL MOVEW$ ENTRY (POINTER, POINTER, FIXED BIN);
CALL MOVEW$ (from_ptr, to_ptr, num_halfwords);
```

Parameters

from_ptr

INPUT. Pointer to place to move from.

to_ptr

INPUT. Pointer to place to move to.

num_halfwords

INPUT. Number of halfwords to move. A halfword is 16 bits.

Discussion

Make sure that the two areas of memory you are using do not overlap

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.

NAMEQ$

Purpose

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

Usage

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

eqnam = NAMEQ$ (string1, len1, string2, len2);
```

Parameters

string1

INPUT. The first string for comparison.

len1

INPUT. The length in characters of string1.

string2

INPUT. The second string for comparison.

len2

INPUT. The length in characters of string2.

eqnam

RETURNED VALUE. 1 if the strings are the same, 0 if they are not.

Discussion

NAMEQ$ performs a character-by-character comparison of string1 and string2 for length len1 or len2, whichever is shorter. Then, if the two strings are identical so far and the next character in the longer string is a blank, NAMEQ$ returns 1; if not, it returns 0. For instance, a comparison of HOW and HOW Y returns the value 1 (true), while a comparison of HOW and HOWDY returns 0 (false).

You are likely to need this subroutine only if you are using FORTRAN. Other high-level languages have their own facilities for string comparison.

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.

SCHAR

Purpose

SCHAR stores a character into an array location. Its counterpart is GCHAR, which retrieves a character from an array. GCHAR is described earlier in this section.

Since SCHAR is strictly a FORTRAN tool, its Usage description is given in FORTRAN format.

Usage

```
INTEGER*2 array(1), index, char

CALL SCHAR (LOC(array), index, char)
```

Parameters

LOC(array)

INPUT --> OUTPUT. Pointer to the array of characters in which the character is to be stored.

index

INPUT/OUTPUT. Index of the location of char in the array. Incremented by 1 after each call to SCHAR.

char

INPUT. Character to be stored. It must be in the right-hand byte of a 16-bit integer.

Discussion

SCHAR is helpful for storing character data from a FORTRAN program.

If you are storing characters starting with the beginning of an array, the pointer index, index, must be initialized to 0. It is incremented by 1 after each call to SCHAR. If you are not storing the character in the first position in the array, then you must load index with position (X - 1) in order to store the character at position X.

The right side of char holds the character for storage. For example, to store the single character A you load char with A -- A in the right side of the halfword and the blank character (or any other character) in the left side of the halfword.

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.

Note

Make sure that FORTRAN_IO_LIBRARY.RUN is specified in your search rules.

UID$BT

Purpose

Returns a unique bit string for identification purposes.

Usage

```
DCL UID$BT ENTRY (BIT (48) ALIGNED);

CALL UID$BT (unique_bit_string);
```

Parameters

unique_bit_string

OUTPUT. Unique bit string returned.

Discussion

The string is guaranteed to be unique. This bit string is not random; it is formed by concatenating the current date and time, in FS (File System) format, with a 16-bit counter. (The format of a 32-bit encoded FS-format date is described in Appendix C.) If a random number is required rather than a unique identifier, the applications library routine RAND$A should be 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: Not available.

UID$CH

Purpose

Given a unique bit string, returns a unique character string based on the bit string. This string can be used as a filename.

Usage

```
DCL UID$CH ENTRY (BIT (48) ALIGNED, CHAR (13));

CALL UID$CH (unique_bit_string, character_string);
```

Parameters

unique_bit_string

INPUT. Unique bit string, preferably generated by UID$BT (see UID$BT above).

character_string

OUTPUT. The resulting character string. The string is formed by converting each 4-bit chunk of the bit string into one of 16 consonants and prefixing the result with a $.

Discussion

UID$CH is designed to be used with bit strings generated by UID$BT. See UID$BT for details.

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
Condition Mechanism

This chapter describes subroutines used in the implementation of the condition mechanism. The first part of this chapter describes routines used to signal and catch conditions. The second part of this chapter describes three routines used to control automatic signalling of the EXIT$ condition. The third part of this chapter describes the data structure formats associated with the condition mechanism. Most programs do not use these data structures.

For a list of the standard conditions raised by Prime software, see Appendix A.

A condition is an unscheduled software procedure call (or block activation) resulting from an "unusual event." Such an unusual event might be a hardware-defined fault, an error situation that cannot be adequately handled in the current subroutine, or an external event such as a QUIT from the user terminal. The condition mechanism

- Provides a consistent and useful means for system software to handle error conditions.
- Provides the capability for programs to handle error conditions without forcing a return to command level.
- Provides support for the condition mechanism of ANSI PL/I.

When such an event happens, PRIMOS is asked to find a handler (an on-unit). PRIMOS accomplishes this by searching the process's stack for frames that have predefined on-units that can handle that named

condition. If PRIMOS finds a handler, the handler is invoked and, upon return from the handler, one of three things can occur:

- Execution can return to the program at the point the event was encountered, and the program then continues.
- Execution can return to the program at some other point, using the mechanism called "nonlocal GOTO."
- The system can continue to search for other handlers for the named event.

The on-unit controls which of these three paths is taken.

The subroutines described in this chapter allow the programmer to create and use on-units. These features are available to programmers using all Prime-supplied languages. The descriptions below use mostly PL/I terminology, with special advice for FORTRAN users.

Appendix A contains a list of system-defined conditions. Because PRIMOS error handling uses conditions, the list of condition names is helpful in interpreting error messages printed by PRIMOS.

CREATING AND USING ON-UNITS

Condition handlers are called on-units. They can be procedures or PL/I begin blocks. A begin block results from a PL/I ON statement, but a procedure results from the use of the following subroutines:

MKONU$

MKON$F

MKON$P

The use of these subroutines is the only way to create an on-unit in a non-PL/I environment. See Table 7-1 to determine which subroutine to use.

The correct on-unit is found by searching backwards through the call stack until an appropriate procedure activation is encountered. A appropriate procedure activation is one that has previously created an on-unit for the condition. If none is found, but if an on-unit for the special condition ANY$ does exist, the ANY$ on-unit is selected as the default on-unit.

All users are automatically protected by PRIMOS, which catches all conditions as a last resort and takes appropriate default action.

Table 7-1
Subroutines Appropriate to Various Languages

Action	Programming Language(1) FTN	F77, C, Pascal	PL/I	PMA
Create an on-unit	MKON$F	MKON$P	MKON$P(2)	MKONU$(3)
Signal a condition	SGNL$F	SGNL$F	SIGNL$	SIGNL$
Cancel (revert) an on-unit	RVON$F	RVON$F	RVONU$(4)	RVONU$
Nonlocal GOTO	PL1$NL	PL1$NL	(5)	PL1$NL
Make PL/I-compatible label	MKLB$F	MKLB$F	(5)	MKLB$F

Notes to Table 7-1

1. The CPL language, not shown in this table, also supports the condition mechanism, but without the use of these subroutine calls. See EXAMPLES OF PROGRAMS later in this chapter.

2. MKON$P is required for programmer-named conditions. Several predefined conditions are supported by the language's ON statement. It is also possible to use MKONU$ instead of MKON$P. See MKONU$ under CONDITION MECHANISM ROUTINES later in this chapter.

3. You must provide an extended stack area, and, while the condition handler is active, you must not modify the character-varying variable that holds the condition name.

4. Use the language-supplied REVERT statement for PL/I predefined conditions.

5. Supported directly by the programming language.

An on-unit can be invalidated by the PL/I REVERT statement or by using the following subroutines:

RVONU$

RVON$F

Again, use Table 7-1 to select the proper subroutine.

The condition mechanism is activated whenever a condition is raised. A condition is raised implicitly by some exception being detected during regular program execution. A condition may be raised explicitly by the PL/I signal statement or by a call to the following subroutines:

SIGNL$

SGNL$F

Every on-unit has the name of the condition it is handling. A condition name is a character string (up to 32 characters) and may represent a system-defined condition if the name is one reserved for system use. If the name is not one reserved for system use, the on-unit represents a user-defined condition. The system-defined conditions are described in Appendix A.

It is extremely important that all on-unit procedures take at least one argument.

On-unit Actions

An on-unit has several options for action it may take. An on-unit may

- Perform application-specific tasks (such as closing or updating files).
- Repair the cause of the condition and then resume execution.
- Decide that normal flow can be interrupted and that the program can be reentered at a "known point" by performing a nonlocal GOTO to some previously defined label.
- Signal another condition.
- Transfer the process to the command level.
- Continue to search for more on-units.
- Run diagnostic routines.

FORTRAN Considerations

The use of on-units and of nonlocal GOTOs is somewhat restricted from FORTRAN, since there are no internal procedures or blocks. Therefore,

- FORTRAN on-units must be subroutines that, by definition, are not internal to the subroutine or main program creating the on-unit.
- Nonlocal GOTOs work only to a previous stack level since the target statement label belongs to the caller of the subroutine performing the nonlocal GOTO.

A full-function nonlocal GOTO requires that the target label identify both a statement and a stack frame of the program that contains the statement. The subroutine MKLB$F creates a PL/I-compatible label and the subroutine PL1$NL performs a nonlocal GOTO to a specified target label. Labels produced by MKLB$F are acceptable to PL1$NL.

This chapter documents subroutines in PL/I notation. FORTRAN users can convert between PL/I and FORTRAN data types by using Table 7-2.

Table 7-2
Conversion of PL/I to FORTRAN Data Types

PL/I	FORTRAN
char(n)	INTEGER((n+1)/2)
char(n) var	INTEGER(((n+1)/2)+1)
fixed bin(15)	INTEGER*2
fixed bin(31)	INTEGER*4
label	REAL*8
entry variable	REAL*8
ptr options (short)	INTEGER*4
bit(n)	INTEGER*2 (1<=n<=16)

The PL/I interfaces use the PL/I data type "character(*) varying", which is not available in FTN. However, 1977 ANSI FORTRAN (F77) includes the data type "character*n", which is the equivalent of PL/I "character(n), nonvarying". Interfaces are provided that use the nonvarying character strings. It is possible to simulate varying character strings in FORTRAN with an INTEGER*2 array in which the first element contains the character count and the remaining elements contain the characters in packed format. For example:

```
PL/I
    DCL NAME CHAR(5) VARYING STATIC INITIAL ('QUIT$');

FORTRAN
    INTEGER*2 NAME(4)
    DATA NAME/5, 'QUIT$'/
```

For information on mapping PL/I data types to other languages, such as Pascal, COBOL, and C, see Volume I of the Subroutines Reference Guide.

On-units must be carefully designed not to require reentrancy which is not supported by FORTRAN. See how I/O must be handled in EXAMPLES OF PROGRAMS, below.

Default On-unit

The default on-unit, ANY$, can be created to intercept any condition that might be activated during a procedure. (The ANY$ on-unit is created by a call to MKONU$ or MKON$F.)

When a condition is raised, the condition mechanism first searches for an on-unit for the specific condition. If a specific on-unit exists, it is selected. Otherwise, if an ANY$ on-unit exists, the ANY$ on-unit is selected.

Your programs should avoid the use of the ANY$ on-unit. Your ANY$ on-unit should not attempt to handle most system-defined conditions, but should pass them on to the next on-unit by simply returning. Whenever an ANY$ on-unit is invoked, the continue switch is set and your ANY$ on-unit must return with the continue switch still set. Failure to do so can cause problems with PRIMOS.

The continue switch indicates to the condition mechanism whether the on-unit that was just invoked (or any of its dynamic descendants) wishes the backward scan of the stack for on-units for this condition to continue upon the on-unit's return. The subroutine CNSIG$ is used to request that the switch be turned on. This switch is cleared before

each on-unit (except ANY$) is invoked. See the discussion of the continue switch at cflags.continue_sw under DATA STRUCTURE FORMATS later in this chapter.

Note

The Prime Symbolic Debugger (DBG) uses the standard condition ILLEGAL_INST$ internally. If you create an on-unit for ILLEGAL_INST$, or if an on-unit for ANY$ handles the ILLEGAL_INST$ condition, such an on-unit must continue the signal if the program is to be successfully debugged using DBG.

EXAMPLES OF PROGRAMS

Below are sample programs in FORTRAN 66 (FTN), FORTRAN 77 (F77), PL/I (PL1), and CPL that use an on-unit to trap the QUIT$ condition. The programs are similar, but not identical, in operation.

Note

In both FORTRAN examples (FTN and F77), the on-unit must avoid using standard FORTRAN I/O, and instead uses TNOU. The condition has arisen in the middle of FORTRAN input, and since FORTRAN I/O is not reentrant, use of FORTRAN I/O by the on-unit would destroy the environment to which it eventually returns. PL/I supports reentrancy and does not require this precaution.

FORTRAN Example

```
C Program to demonstrate on-unit in FTN
C
      EXTERNAL CATCH
      INTEGER*2 BREAK(3), BREAKL, I
      DATA BREAK/'QUIT$'/
      BREAKL = 5
      CALL MKON$F(BREAK, BREAKL, CATCH)
      WRITE(1,300)
300   FORMAT('Please enter an integer, then RETURN.')
100   CONTINUE
      READ(1,200) I
200   FORMAT(I8)
      IF (I .EQ. 0) GOTO 400
      WRITE(1,330)
330   FORMAT('Again, 0 to exit, BREAK to test on-unit.')
      GOTO 100
400   STOP
      END
```

```
C
      SUBROUTINE CATCH(PNTR)
      INTEGER*4 PNTR
      CALL TNOU('We caught a quit!',17)
      PAUSE 1
      CALL TNOU('You''re back into the input loop again.',38)
      RETURN
      END
```

FORTRAN 77 Example

```
C Program to demonstrate on-unit in F77
C
      external catchit
      integer*2 break_length
      character*5 break/'QUIT$'/
      break_length = 5
      call mkon$p(break,break_length,catchit)
      print*, 'Please enter an integer, then RETURN.'
100   continue
      read(1,*) i
      if (i.eq.0) goto 200
      print*, 'Again, 0 to exit, BREAK to test on-unit.'
      goto 100
200   end
      subroutine catchit(pntr)
      integer*4 pntr
      call tnou('We caught a quit!',ints(17))
      pause 1
      call tnou('You''re back into the input loop again.',ints(38))
      return
      end
```

PL/I Examples

```
/* Program to demonstrate on-unit in PL1 */

ex_pl1: procedure options (main);
   dcl mkon$p entry(char(*), fixed bin, entry);
   dcl (break_length, i) fixed bin(15);
   dcl (break) character(5) static initial('QUIT$');
   break_length = 5;
   call mkon$p(break, break_length, catchit);
   put skip list ('Please enter an integer, then RETURN.');
   get list (i);
   do while (i ^= 0);
      put skip list ('Again, 0 to exit, BREAK to test on-unit.');
      get list (i);
   end;
   stop;
```

```
   catchit: proc (pntr);
      dcl pntr pointer;
      put skip list ('We caught a quit!');
      put skip list('You''re back into the input loop again.');
      return;
   end;
end;
```

```
 /* Modified program to demonstrate on-unit in PL1 */
 /* Shows use of MKONU$ (instead of MKON$P) */

ex_pl1: procedure options (main);
   declare mkonu$ entry(character(32) varying, entry)
                  options(shortcall(20));
   declare (break) character(32) static initial('QUIT$') varying;
   declare i fixed binary(15);
   call mkonu$(break, catchit);
   put skip list ('Please enter an integer, then RETURN.');
   get list (i);
   do while (i ^= 0);
      put skip list ('Again, 0 to exit, BREAK to test on-unit.');
      get list (i);
   end;
   stop;

   catchit: procedure (pntr);
      declare pntr pointer;
      put skip list ('We caught a quit!');
      put skip list('You''re back into the input loop again.');
      return;
   end;
end;
```

CPL Example

```
/* Program to demonstrate on-unit in CPL.
/* Note that CPL cannot call a make-on-unit
/* subroutine.  Instead, we show the use of
/* the ON statement provided by CPL.

&on QUIT$ &routine catchit
type 'Please enter an integer, then RETURN.'
&set_var i := [response '']
&do &while %i% ^= 0
   type 'Again, 0 to exit, BREAK to test on-unit.'
   &set_var i := [response '']
&end
&stop
```

```
&routine catchit
type 'We caught a quit!'
type 'You''re back into the input loop again.'
&return
```

ADDITIONAL PROGRAM EXAMPLES

The programs presented below show strategies for using the condition mechanism. The examples include

- CPL programs that handle on-units for a program that does not itself use on-units
- A FORTRAN 77 (F77) program that shows reentering a program with the PRIMOS REN command. The program also shows the use of the nonlocal GOTO
- A FORTRAN 66 (FTN) program that handles QUIT$ and shows the nonlocal GOTO
- A PL/I (PL1) program that handles end of file
- A FORTRAN 66 program that demonstrates the CLEANUP$ condition, which is raised while processing a nonlocal GOT.

Two Protecting Programs in CPL

Below are two programs, each of which protects a FORTRAN program called SQRT against being interrupted by the BREAK (or CONTROL-P) key. They demonstrate both a simple and a more sophisticated means by which programs can avoid having to use the condition mechanism subroutines. When the language in which a program is written does not support on-units, or when condition handling is added as an afterthought, CPL can sometimes be used to handle conditions.

```
/* PROTECT.CPL
/* Trap the BREAK key with an on-unit in CPL.
/*
&ON QUIT$ &ROUTINE BREAK_HANDLER
&DATA SEG SQRT
  &TTY
&END
&RETURN
```

```
&ROUTINE BREAK_HANDLER
  TYPE
  TYPE
  TYPE You have typed the break key.
  &SET_VAR EXIT_FLAG := ~
    [QUERY 'Do you wish to exit from the program']
  &IF ^ %EXIT_FLAG% ~
  &THEN ~
    TYPE Continuing program.
  &ELSE ~
    &DO
      TYPE Exiting program.
      &STOP
    &END
  &RETURN
```

The program PROTECT2.CPL can better handle your typing BREAK several times in a row.

```
/* PROTECT2.CPL
/* Trap the BREAK key with an on-unit in CPL.
/* Do not allow multiple breaks.
/*
&ON QUIT$ &ROUTINE BREAK_HANDLER
&DATA SEG SQRT
  &TTY
&END
&RETURN

&ROUTINE BREAK_HANDLER
  &ON QUIT$ &ROUTINE DUMMY_HANDLER
  TYPE
  TYPE
  TYPE You have typed the break key.
  &LABEL ALTERNATE_ENTRY
  &SET_VAR EXIT_FLAG := ~
    [QUERY 'Do you wish to exit from the program']
  &IF ^ %EXIT_FLAG% ~
  &THEN ~
    TYPE Continuing program.
  &ELSE ~
    &DO
      TYPE Exiting program.
      &STOP
    &END
  &RETURN

&ROUTINE DUMMY_HANDLER
  TYPE
  TYPE Please answer the question!
  &GOTO ALTERNATE_ENTRY
  &RETURN
```

Here is the FORTRAN source for the SQRT program invoked by PROTECT and PROTECT2.

```
C  SQRT.FTN
C
C  This is a small interactive FORTRAN program that is to be
C  protected from BREAKs (the QUIT$ condition) by an enveloping
C  program written in CPL.
C
      REAL INVAL, OUTVAL
C
1000  WRITE (1, 1005)
1005  FORMAT (/, 'WHAT IS THE NUMBER:')
      READ (1, 1010) INVAL
1010  FORMAT (F5.0)
      IF (INVAL .EQ. 0.) GOTO 9999
      OUTVAL = SQRT (INVAL)
      WRITE (1, 1020) INVAL, OUTVAL
1020  FORMAT ('THE SQUARE ROOT OF ', F5.0, ' IS ', F5.2)
      GOTO 1000
C
9999  WRITE (1, 9000)
9000  FORMAT (/ , 'END OF PROGRAM')
      CALL EXIT
      END
```

The REENTER$ Condition From F77

```
C  REENTER.F77
C
C  This program creates an on-unit for the REENTER$ condition.
C  If the user breaks out of the program during its operation, and
C  then reenters it through the PRIMOS REN command, the on-unit
C  is invoked to start the program from the proper place.
C
      EXTERNAL RENHDLR
      EXTERNAL MKON$P
      EXTERNAL MKLB$F
C
      CHARACTER*8 CONDITION_NAME/'REENTER$'/
      CHARACTER*80 CHAR_STRING
      REAL*8 REENTRY_POINT
      INTEGER*2 INDEX, CONDITION_LENGTH/8/
C
      COMMON /REENTRY/ REENTRY_POINT
C
C  The "$1000" on the next line refers to statement 1000
      CALL MKLB$F ($1000, REENTRY_POINT)
      CALL MKON$P (CONDITION_NAME, CONDITION_LENGTH, RENHDLR)
C
```

```
1000  WRITE (1, 1010)
1010  FORMAT ('Enter a character string:')
      READ (1, 1020) CHAR_STRING
1020  FORMAT (A80)
C
      DO 9999 INDEX = 1, 500
      WRITE (1, 1030) CHAR_STRING
1030  FORMAT (A80)
9999  CONTINUE
      END
C
C
      SUBROUTINE RENHDLR (CP)
C
      INTEGER*4 CP
C
      EXTERNAL PL1$NL
      COMMON /REENTRY/ REENTRY_POINT
      WRITE (1, 1010)
1010  FORMAT ('** Reentering subsystem **')
      CALL PL1$NL (REENTRY_POINT)
      RETURN
      END
```

Handling QUIT$ from FTN

```
C  PROSQRT.FTN
C
C  This program creates an on-unit for the BREAK key.  The on-unit
C  prevents BREAK from exiting the program and instructs the user
C  how to exit.
C
C  In FTN the on-unit must be declared as an external routine.
C
      EXTERNAL BKHNDL
C
      REAL INVAL, OUTVAL
      REAL*8 BRKRTN
C
      COMMON /BRKLBL/ BRKRTN
C
      CALL MKON$F ('QUIT$', 5, BKHNDL)
C  The "$1000" in the next line refers to statement 1000
      CALL MKLB$F ($1000, BRKRTN)
1000  WRITE (1, 1005)
1005  FORMAT (/, 'WHAT IS THE NUMBER:')
      READ (1, 1010) INVAL
1010  FORMAT (F5.0)
      IF (INVAL .EQ. 0.) GOTO 9999
      OUTVAL = SQRT (INVAL)
      WRITE (1, 1020) INVAL, OUTVAL
1020  FORMAT ('THE SQUARE ROOT OF ', F5.0, ' IS ', F5.2)
```

```
      GOTO 1000
C
9999  WRITE (1, 9000)
9000  FORMAT (/ , 'END OF PROGRAM')
      CALL EXIT
      END

C
C  This subroutine handles the QUIT$ condition when it is raised.
C
C  Ordinarily, it would be incorrect to use FORTRAN I/O from inside
C  this on-unit, because FTN is not reentrant, and we would be
C  disturbing the keyboard I/O that was in progress when QUIT$
C  was raised.  In this case, however, we use a nonlocal GOTO to
C  return to statement 1000 of the main program, and never return
C  to the I/O that was in progress.
C
      SUBROUTINE BKHNDL (CP)
C
      INTEGER*4 CP
      REAL*8 BRKRTN
      COMMON /BRKLBL/ BRKRTN
      WRITE (1, 1000)
1000  FORMAT ('YOU MUST TYPE ZERO TO EXIT THIS PROGRAM!')
      CALL PL1$NL (BRKRTN)
      RETURN
      END
```

Handling End of File From PL/I

```
/*  EOF.PL1  */

/*  This program creates on-units for both the ENDFILE and QUIT$
    conditions.  The on-unit for the end-of-file condition is
    set up by PL/I's ON statement, while the on-unit for quits
    is set up by calling MKON$P.  The on-unit for quits closes
    all files and exits the program.
 */
EXAMPLE: PROCEDURE OPTIONS(MAIN);

    DCL EMPLOYEE_NO FIXED DECIMAL(5);
    DCL (GROSS_PAY, HOURLY_RATE) FIXED DECIMAL(5,2);
    DCL HOURS_WORKED FIXED DECIMAL(2);
    DCL FIXED DECIMAL(5,2);
    DCL NUMBER_OF_EMPLOYEES FIXED BIN(15);
    DCL (REPORT, DATAFILE) FILE;
    DCL CONDITION_NAME CHAR(5) STATIC INITIAL('QUIT$');
    DCL MKON$P ENTRY (CHAR(5), FIXED BIN, ENTRY);

    BREAK_HANDLER: PROC(CP);
        DCL CP PTR;
        PUT SKIP LIST ('** Aborting program **');
```

```
        CLOSE FILE (DATAFILE);
        CLOSE FILE (REPORT);
        GOTO ABORT_PROGRAM;
    END;

    ON ENDFILE (DATAFILE)
    BEGIN;
        PUT SKIP LIST ('** End of File Encountered **');
        GOTO END_FILE;
    END;

    CALL MKON$P (CONDITION_NAME, 5, BREAK_HANDLER);
    OPEN FILE (DATAFILE) TITLE ('DATAFILE') STREAM INPUT;
    OPEN FILE (REPORT) TITLE  ('REPORT')  STREAM OUTPUT;
    NUMBER_OF_EMPLOYEES = 0;

    DO WHILE('1'B);
        GET FILE (DATAFILE)
            LIST (EMPLOYEE_NO, HOURLY_RATE, HOURS_WORKED);
        NUMBER_OF_EMPLOYEES = NUMBER_OF_EMPLOYEES + 1;
        GROSS_PAY = HOURS_WORKED * HOURLY_RATE;
        PUT FILE (REPORT)
         LIST (EMPLOYEE_NO, HOURLY_RATE,
               HOURS_WORKED, GROSS_PAY);
        PUT FILE(REPORT)SKIP;
    END;

    END_FILE:
    PUT FILE(REPORT) LIST(NUMBER_OF_EMPLOYEES)SKIP(3);

    ABORT_PROGRAM:
END EXAMPLE;
```

A CLEANUP$ On-unit From FTN

The following programs demonstrate the QUIT$ and CLEANUP$ on-units. When the BREAK key is typed, a nonlocal GOTO is executed, which causes CLEANUP$ to be raised in the routine SUBA.

```
C  CLEANUP.FTN
C
C  This program creates on-units for the QUIT$ and CLEANUP$
C  conditions.
C
      EXTERNAL BKHNDL
C
      REAL*8 BRKRTN
      COMMON /BRKLBL/ BRKRTN
C
      CALL MKON$F ('QUIT$', 5, BKHNDL)
      CALL MKLB$F ($1000, BRKRTN)
1000  WRITE (1,1010)
```

```
1010  FORMAT (/, 'In the routine: MAIN')
      CALL SUBA
      CALL EXIT
      END
C
      SUBROUTINE SUBA
      EXTERNAL ACLUP
      WRITE (1, 1000)
1000  FORMAT ('In the routine: SUBA')
      CALL MKON$F ('CLEANUP$', 8, ACLUP)
      CALL SUBB
      RETURN
      END
C
      SUBROUTINE SUBB
      INTEGER DUMMY
      WRITE (1,1000)
1000  FORMAT ('In the routine: SUBB')
      WRITE (1, 1010)
1010  FORMAT ('Type RETURN to exit, BREAK to test on-units')
      READ (1, 1020) DUMMY
1020  FORMAT (A2)
      RETURN
      END


C  HDLRS.FTN
C
C  On-units for the module CLEANUP.FTN
C
C  The routine ACLUP is invoked when a nonlocal GOTO is
C  aborting SUBA.
C
      SUBROUTINE ACLUP (CP)
      INTEGER*4 CP, I
      WRITE (1, 1000)
1000  FORMAT ('In the cleanup routine: ACLUP')
      DO 1010 I = 1, 50000
1010  CONTINUE
      RETURN
      END
C
C  The routine BKHNDL is invoked when the QUIT$ condition is
C  raised by the user hitting the BREAK key.
C
      SUBROUTINE BKHNDL (CP)
      INTEGER*4 CP
      REAL*8 BRKRTN
      COMMON /BRKLBL/  BRKRTN
      WRITE (1, 1000)
1000  FORMAT ('In the routine: BKHNDL')
      CALL PL1$NL (BRKRTN)
      RETURN
      END
```

CRAWLOUT MECHANISM

An event known as a crawlout occurs whenever the condition mechanism reaches the end of an inner-ring stack (a ring other than ring 3) without finding a selectable on-unit for the condition that has been raised. (Protection rings are described in the System Architecture Reference Guide.) A crawlout can occur even when the inner ring has an on-unit for the condition. This occurs if that on-unit signals another condition, or calls CNSIG$ and returns, causing a resumption of the stack scan. The scan for on-units resumes on the stack of the ring that invoked the inner ring. The outer ring receives a copy of the machine state at the time the condition was raised.

CONDITION MECHANISM ROUTINES

This section describes the following subroutines:

Routine	Function
CNSIG$	Continues scan for on-units.
MKLB$F	Converts FORTRAN statement label to PL/I format.
MKON$F	Creates an on-unit (for FTN users).
MKON$P	Creates an on-unit (for any language except FTN).
MKONU$	Creates an on-unit (for PMA and PL/I users).
PL1$NL	Performs a nonlocal GOTO.
RVON$F	Reverts an on-unit (for FTN users).
RVONU$	Reverts an on-unit (for any language except FTN).
SGNL$F	Signals a condition (for FTN users).
SIGNL$	Signals a condition (for any language except FTN).

CNSIG$

Purpose

CNSIG$ is called when an on-unit has been unable to completely handle the condition. CNSIG$ instructs the condition mechanism to continue scanning for more on-units for the specific condition that was raised after the calling on-unit returns. The continue-to-signal switch, cfh.cflags.continue_sw, is set in the most recent condition frame.

Usage

```
DCL CNSIG$ ENTRY (FIXED BIN);

CALL CNSIG$ (code);
```

Parameters

code

OUTPUT. Standard error code. Nonzero only if there was no condition frame found in the stack.

Discussion

The continue-to-signal switch is automatically set whenever an ANY$ on-unit is invoked. Therefore, an ANY$ on-unit need not issue a call to CNSIG$ to continue to signal.

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.

MKLB$F

Purpose

MKLB$F converts a FORTRAN statement label or an integer variable with a statement label value into a PL/I-compatible label value. This label value can then be used with a call to the subroutine PL1$NL to perform a full-function nonlocal GOTO in a FORTRAN program.

Usage

The FORTRAN usage is:

```
INTEGER*2 stmt
REAL*8 label

CALL MKLB$F (stmt, label)
```

Parameters

stmt

INPUT. Variable to which a FORTRAN statement number has been assigned by an ASSIGN statement, or a statement number constant in the format $xxxxx.

label

OUTPUT. Contains PL/I-compatible label value for stmt returned by call to MKLB$F.

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.

MKON$F

Purpose

MKON$F creates an on-unit for a specific condition and is intended for the FTN user.

Usage

The FORTRAN usage is:

```
EXTERNAL unit
INTEGER*2 cname(16), cnamel

CALL MKON$F (cname, cname_len, unit)
```

Parameters

cname

INPUT. Array containing name of condition for which on-unit is to be created.

cname_len

INPUT. Length (in characters) of cname.

unit

INPUT. The external subroutine that is to be the on-unit handler. The subroutine must take an argument, since the PRIMOS condition mechanism calls the subroutine as follows:

```
INTEGER*4 CP
CALL UNIT (CP)
```

where CP is a pointer to the condition frame header (CFH) that describes the condition.

Discussion

FORTRAN cannot directly access the CFH through CP. A subroutine written in PL/I or PMA could pass the desired CFH information, or the MOVEW$ procedure could be used to move the data to an accessible location.

cname and cname_len can be overwritten by the caller once MKON$F has returned, since they are copied into a stack frame extension.

> **Caution**
>
> MKON$F should not be called from FORTRAN 77. FORTRAN 77 requires MKON$P.

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.

MKON$P

Purpose

MKON$P creates an on-unit for a given condition. It can be used in programs written in any language except FTN.

Usage

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

CALL MKON$P (condname, namelen, handler);
```

Parameters

condname

INPUT. The name of the condition for which an on-unit is desired. The name should not contain any blanks.

namelen

INPUT. The length of condname, in characters.

handler

INPUT. The internal or external entry (subroutine) value that is to be invoked as the on-unit. If the value is an internal procedure, it must be immediately contained in the block calling MKON$P. The subroutine must take at least one argument.

The F77 usage is:

```
EXTERNAL handler
INTEGER*2 namelen
CHARACTER*namelen name/'condname'/

CALL MKON$P(name, namelen, handler)
```

condname

INPUT. The name of the condition for which an on-unit is desired. The name should not contain any blanks (input).

name

INPUT. A variable to hold condname. Its value should not be altered while the condition is active.

namelen

INPUT. The length of condname, in characters.

handler

INPUT. The name of the external subroutine that is to become the on-unit. This subroutine must take at least one argument.

Discussion

An on-unit for the specified named condition is created for the calling block. If the block already has an on-unit for that condition, the on-unit is redefined.

> **Caution**
>
> MKON$P cannot be called from FORTRAN (FTN). FORTRAN requires MKON$F.

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.

MKONU$

Purpose

MKONU$ creates an on-unit for a specific condition or creates a default on-unit for the ANY$ condition. MKONU$ can be called only from PMA and PL/I. PL/I programmers may use either MKON$P or MKONU$.

Usage

```
DCL MKONU$ ENTRY (CHAR(*)VAR, ENTRY) OPTIONS(SHORTCALL (20));

CALL MKONU$ (condition_name, handler);
```

Parameters

condition_name

INPUT. Name of condition for which on-unit will be created. The name cannot contain trailing blanks. Any previous on-unit for this condition within the activation will be overwritten.

handler

INPUT. Entry value representing on-unit procedure to be invoked when condition_name is raised and this activation is reached in the stack scan. Since MKONU$ does not save the display pointer associated with on-unit entry, the entry value must be external or declared in the block calling MKONU$. (An entry constant declared in the block containing the call to MKONU$ satisfies these restrictions.) The handler must take at least one argument.

Discussion

The stack frame of the caller is lengthened, if necessary, to add the descriptor block for the new on-unit.

The caller must guarantee that the storage occupied by condition_name will not be freed until the caller returns or until the activation is aborted by a nonlocal GOTO. The suggested way of making this guarantee is to declare a static character varying field containing the name of the condition, and to use that field in the call.

From PL/I the declaration OPTIONS (SHORTCALL(20)) is required for MKONU$. The PL/I SHORTCALL option provides additional space needed for the calling procedure's temporary storage. OPTIONS(SHORTCALL) provides

8 halfwords of stack by default. MKONU$ requires 28 halfwords of stack, and thus requires an extra 20 halfwords. If the stack size is insufficient, the return from MKONU$ causes unpredictable results.

OPTIONS(SHORTCALL) causes the PMA instruction JSXB to be used instead of the PCL instruction. PCL generates a new stack. JSXB does not generate a new stack, and is faster, but requires that there be sufficient space on the caller's stack. Also, MKONU$ can only be called from code executing in V-mode.

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.

PL1$NL

Purpose

PL1$NL performs a full-function nonlocal GOTO to the statement identified in the call. Label values created by MKLB$F are suitable arguments for PL1$NL.

Usage

The FORTRAN usage is:

```
REAL*8 label

CALL PL1$NL (label)
```

Parameters

label

INPUT. PL/I-compatible label value.

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.

RVON$F

Purpose

RVON$F disables (reverts) an on-unit for a specific condition. Its effect is identical to RVONU$ but is designed for the FTN user.

Usage

The FORTRAN usage is:

```
INTEGER*2 cname(16), cnamel

CALL RVON$F (cname, cnamel)
```

Parameters

cname

INPUT. Name of condition for which the on-unit is to be disabled.

cnamel

INPUT. Length (in characters) of cname.

Discussion

There is no effect if an on-unit does not exist for the named condition, or if the on-unit has already been disabled.

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.

RVONU$

Purpose

RVONU$ disables (reverts) an on-unit for a specific condition for any language except FTN. Once disabled, the on-unit is ignored during stack frame scanning. The on-unit can be reinstated only by another call to MKONU$ or MKON$F. A call to RVONU$ affects only on-units within its own activation. RVONU$ is used from programs written in languages that support the CHARACTER VARYING data type.

Usage

```
DCL RVONU$ ENTRY (CHAR(32) VAR);

CALL RVONU$ (condition_name);
```

Parameters

condition_name

INPUT. Name of condition for which the on-unit is to be disabled.

Discussion

There is no effect if an on-unit does not exist for the named condition, or if the on-unit has already been disabled. A call to RVONU$ does not affect on-units in any other activation.

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.

SGNL$F

Purpose

SGNL$F signals a specific condition and supplies optional auxiliary information. SGNL$F is the FTN equivalent of SIGNL$. It is used from programs written in languages that do not support the CHARACTER VARYING data type.

Usage

The FORTRAN usage is:

```
INTEGER*2 cname(16), cnamel, mslen, infoln, flags
INTEGER*4 msptr, infopt

CALL SGNL$F (cname, cnamel, msptr, mslen, infopt, infoln, flags);
```

Parameters

cname

INPUT. Name of condition to be signalled.

cnamel

INPUT. Length (in characters) of cname.

msptr

INPUT. Pointer to location of stack frame header describing machine state at time the specific condition was detected. The user does not usually know this information and should pass the null pointer value (:177600000).

mslen

INPUT. Length (in halfwords) of stack frame header.

infopt

INPUT. Pointer to location of user-supplied auxiliary information array. If no information is supplied, the user should pass the null pointer value (:1777600000).

infoln

INPUT. Length (in halfwords) of the structure pointed to by infopt.

flags

INPUT. Flag array specifying control action:

Bit	Meaning
1	If =1, on-unit may return.
2	If =1, on-unit may return without taking action.
3	If =1, call is result of crawlout. This bit should never be set by the user.
4	If =1, signal PL/I I/O (PLIO) condition. User program should not set.
5-16	Must be 0.

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.

SIGNL$

Purpose

SIGNL$ is called to signal a specific condition. The stack is scanned backwards to find an on-unit for this condition or a default (ANY$) on-unit. SIGNL$ is used for any language except FTN.

Usage

```
DCL SIGNL$ ENTRY (CHAR(*) VAR, PTR, FIXED BIN, PTR, FIXED BIN,
                  BIT(16) ALIGNED);

CALL SIGNL$ (condition_name, ms_ptr, ms_len, info_ptr,
             info_len, action);
```

Parameters

condition_name

INPUT. Name of condition to be signalled.

ms_ptr

INPUT. Pointer to stack frame header structure defining the machine state at the time the specific condition was detected. If ms_ptr is null, a pointer to the condition frame header produced by this call to SIGNL$ is used.

ms_len

INPUT. Length (in halfwords) of the structure named in ms_ptr. It is not examined if ms_ptr is null.

info_ptr

INPUT. Pointer to structure containing auxiliary information about the condition. If no auxiliary information is available, info_ptr should be null.

info_len

INPUT. Length (in halfwords) of structure in info_ptr. It is not examined if info_ptr is null.

action

INPUT. A 16-bit halfword that defines action to be taken:

```
DCL 1 action,
     2 return_ok bit(1),
     2 inaction_ok bit(1),
     2 crawlout bit(1),
     2 specifier bit(1),
     2 mbz bit(12);
```

return_ok If = '1'b, on-unit is to be allowed to return.

inaction_ok If = '1'b, on-unit may return without taking corrective action and still expect "defined" results. (return_ok must also be '1'b.)

crawlout If = '1'b, call to SIGNL$ is result of a crawlout. It should never be set by user.

specifier If = '1'b, it signals PL/I I/O (PLIO) condition. User program should not use.

mbz Must be 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.

EXIT CONDITION CONTROL ROUTINES

This section describes the following subroutines:

Routine	Function
EX$CLR	Disables signalling of EXIT$ condition.
EX$RD	Returns state of EXIT$ signalling.
EX$SET	Enables signalling of EXIT$ condition.

EX$CLR

Purpose

This routine disables the signalling of the EXIT$ condition either after a program's completion or after its termination as the result of a nonlocal GOTO having been executed.

Usage

```
DCL EX$CLR ENTRY ();

CALL EX$CLR;
```

Parameters

There are no parameters.

Discussion

To disable the EXIT$ condition, one call to EX$CLR must be made for every call to EX$SET, as PRIMOS looks to a single counter that is either incremented or decremented by calls to these two routines.

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.

EX$RD

Purpose

This routine returns the state of the counter used to control the conditional signalling of the EXIT$ condition whenever a program EPF (executable program format) terminates. The routine EX$SET enables the EXIT$ condition; the routine EX$CLR disables it.

Usage

```
DCL EX$RD ENTRY (FIXED BIN(15));

CALL EX$RD (transmit_exit_setting);
```

Parameters

transmit_exit_setting

OUTPUT. The value returned from the counter. A value greater than zero enables the signalling of the EXIT$ condition whenever a program terminates. If the value is zero or negative, the signal is disabled.

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.

EX$SET

Purpose

This routine enables the signalling of the EXIT$ condition either after a program's completion or after its termination as the result of a nonlocal GOTO having been executed.

Usage

```
DCL EX$SET ENTRY ();

CALL EX$SET;
```

Parameters

There are no parameters.

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.

DATA STRUCTURE FORMATS

The data structures associated with the condition mechanism are described below. Any user program that uses these structures should examine the version number in the structure, if one is provided. If the format of a structure changes, the version number is incremented. The user program can then take appropriate action if it is presented with structures of different formats.

The Condition Frame Header (CFH)

The following declaration shows the format of the standard condition frame header:

```
dcl    1 cfh based,  /* standard condition frame header */
           2 flags,
             3 backup_inh bit(1),
             3 cond_fr bit(1),
             3 cleanup_done bit(1),
             3 efh_present bit(1),
             3 user_proc bit(1),
             3 mbz bit(9),
             3 fault_fr bit(2),
           2 root,
             3 mbz bit(4),
             3 seg_no bit(12),
           2 ret_pb ptr options (short),
           2 ret_sb ptr options (short),
           2 ret_lb ptr options (short),
           2 ret_keys bit(16) aligned,
           2 after_pcl fixed bin,
           2 hdr_reserved(8) fixed bin,
           2 owner_ptr ptr options (short),
           2 cflags,
             3 crawlout bit(1),
             3 continue_sw bit(1),
             3 return_ok bit(1),
             3 inaction_ok bit(1),
             3 specifier bit(1),
             3 mbz bit(11),
           2 version fixed bin,
           2 cond_name_ptr ptr options (short),
           2 ms_ptr ptr options (short),
           2 info_ptr ptr options (short),
           2 ms_len fixed bin,
           2 info_len fixed bin,
           2 saved_cleanup_pb ptr options (short);
```

`flags.backup_inh` Is always '0'b in a condition frame. It is used in regular call frames to control program counter backup on crawlout from an inner ring.

flags.cond_fr	Identifies this frame as a condition frame, and thus is '1'b.
flags.cleanup_done	Is '1'b when this activation has been "cleaned up" by the procedure unwind, which helps to affect nonlocal GOTOs. When this flag is set, the value of cfh.ret_pb no longer describes the return point of the activation; that information is available in cfh.saved_cleanup_pb.
flags.efh_present	Is always '0'b in a condition frame. It is used in a regular call frame to indicate that an extended stack frame header containing on-unit data is present.
flags.user_proc	Identifies stack frames belonging to user or library procedures, and hence is '0'b in a condition frame.
flags.mbz	Reserved and must be '0'b.
flags.fault_fr	Is always '0'b in a condition frame.
root.mbz	Is reserved and must be '0'b.
root.seg_no	Is the hardware-defined stack root segment number, and indicates which segment contains the stack root for the stack containing this fault frame.
ret_pb	Points to the next instruction to be executed following the call to SIGNL$ that caused this condition to be raised, unless flags.cleanup_done is '1'b, in which case cfh.ret_pb points to a special code sequence used during stack unwinds, and cfh.saved_cleanup_pb contains the former value of cfh.ret_pb.
ret_sb	Is the hardware-defined stack base of the caller of SIGNL$. Thus, this value also points to the previous stack frame on the stack.
ret_lb	Is the hardware-defined linkage base of the caller of SIGNL$.
ret_keys	Is the hardware-defined keys register of the caller of SIGNL$.
after_pcl	Is the hardware-defined offset of the first argument pointer following the call to SIGNL$ that raised this condition.
hdr_reserved	Is reserved for future expansion of the hardware-defined PCL/CALF stack frame header, of which the totality of CFH is a further extension.

owner_ptr — Is reserved to point to the entry control block (ECB) of the procedure that owns this stack frame (usually SIGNL$).

cflags.crawlout — If '1'b, this condition occurred in an inner ring (a ring number lower than the ring in which the on-unit is executing), but could not be adequately handled there; otherwise it is '0'b.

cflags.continue_sw — Is used to indicate to the condition mechanism whether the on-unit that was just invoked (or any of its dynamic descendants) wishes the backward scan of the stack for on-units for this condition to continue upon the on-unit's return. The subroutine CNSIG$ is used to request that cflags.continue_sw be turned on; user programs should not attempt to set it directly. This switch is cleared before each on-unit is invoked. ANY$ on-units are exceptions; this switch is set before an ANY$ on-unit is invoked.

cflags.return_ok — If '1'b, indicates the procedure that raised the condition is willing for control to be returned to it by means of the on-unit simply returning. If '0'b, an attempt by an on-unit for this condition to return causes the special condition ILLEGAL_ONUNIT_RETURN$ to be signalled. The on-unit can return regardless of the state of cfh.cflags.return_ok if cfh.cflags.continue_sw has previously been set by a call to CNSIG$. This is because, in this case, the on-unit return does not cause a return to the procedure that raised the condition, but instead causes a resumption of the stack scan.

cflags.inaction_ok — If '1'b, indicates the procedure that raised the condition has determined that it makes sense for an on-unit for this condition to return without taking any corrective action. If '0'b, the on-unit must take some corrective action before returning, or else continued computation may be undefined. cflags.inaction_ok never is '1'b unless cflags.return_ok is '1'b as well. No user program should change the state of this or any other member of cfh.cflags.

cflags.specifier — If '1'b, indicates that this condition is a PL/I I/O (PLIO) condition that requires a specifier pointer, as well as a condition name to completely identify it. This specifier is usually a pointer to a PLIO file control block. The specifier must be the first member of the information structure.

cflags.mbz	Is reserved for future expansion and must be '0'b.
version	Identifies the version number (and hence the format) of this structure, and currently is always 1.
cond_name_ptr	Is a pointer to the name (char(32) varying) of the condition that caused the on-unit to be invoked.
ms_ptr	Is a pointer to a structure that defines the state of the CPU at the time the condition occurred. In the case of hardware faults, ms_ptr points to a standard fault frame header (FFH). In the case of software-initiated conditions, ms_ptr points to a CFH. The two cases can be distinguished by the value of ms_ptr -> cfh.flags.fault_fr. If '0'b, the software case obtains; otherwise, the hardware case obtains.
info_ptr	Is a pointer to an arbitrary structure containing auxiliary information about the condition. If null, no information is available. This pointer is copied directly from the corresponding argument to SIGNL$. If cflags.specifier is '1'b, the format of this structure is partially constrained as described above.
ms_len	Is the length (in halfwords) of the structure pointed to by ms_ptr.
info_len	Is the length (in halfwords) of the structure pointed to by info_ptr.
saved_cleanup_pb	Is valid only if flags.cleanup_done is '1'b, and if valid is the former value of cfh.ret_pb (which has been overwritten by the nonlocal GOTO processor).

Note

When writing procedures to interpret the data contained in a CFH structure, be aware that, in the case of a crawlout, cfh.ms_ptr describes the machine state at the time the condition was generated. The stack history pertaining to that machine state has been lost as a result of the crawlout.

The machine state extant at the time the inner ring was entered is available, and is pointed to by cfh.ret_sb. This machine

state will be a CFH or an FFH according to whether the inner ring was entered via a procedure call (CFH) or a fault (FFH). The value of cfh.ret_sb -> cfh.flags.fault_fr can be used to distinguish these cases.

In the case in which a crawlout has not occurred, cfh.ms_ptr points to the proper machine state, and no assumptions can be made concerning cfh.ret_sb.

For more information on crawlout, see CRAWLOUT MECHANISM earlier in this chapter.

The Extended Stack Frame Header (EFH)

Any procedure (or begin block) that is to create one or more on-units must reserve space in its stack frame header for an extension that contains descriptive information about those on-units. This space is allocated automatically by the Prime high-level language compilers. PMA programs require explicit space allocation. The format of the stack frame header (with extension) is:

```
dcl    1 sfh based,  /* stack frame header */
           2 flags,
             3 backup_inh bit(1),
             3 cond_fr bit(1),
             3 cleanup_done bit(1),
             3 efh_present bit(1),
             3 user_proc bit(1),
             3 stk_cbits bit(1),
             3 lib_proc bit(1),
             3 ecb_cbits bit(1),
             3 mbz bit(6),
             3 fault_fr bit(2),
           2 root,
             3 mbz bit(4),
             3 seg_no bit(12),
           2 ret_pb ptr options (short),
           2 ret_sb ptr options (short),
           2 ret_lb ptr options (short),
           2 ret_keys bit(16) aligned,
           2 after_pcl fixed bin,
           2 hdr_reserved(8) fixed bin,
           2 owner_ptr ptr options (short),
           2 tempsc(8) fixed bin,
           2 onunit_ptr ptr options (short),
           2 cleanup_onunit_ptr ptr options (short),
           2 next_efh ptr options (short),
           2 reserved(6) fixed bin,
           2 cond_bits bit(16) aligned;
```

```
dcl    1 ecb based,  /* Entry Control Block */
          2 pb ptr options (short),
          2 frame_size fixed bin(15),
          2 stack_seg fixed bin(12),
          2 arg_offset fixed bin(15),
          2 num_args fixed bin(15),
          2 lb ptr options (short),
          2 cond_bits bit(16) aligned,
          2 reserved(6) fixed bin(15);
```

flags.backup_inh — Is examined only if this stack frame is the crawlout frame on an inner-ring stack, and a crawlout is taking place. If '1'b, it indicates that sfh.ret_pb is to be copied to the outer ring as-is, so that the operation being aborted by the crawlout is not retried. If '0'b, sfh.ret_pb is set to point at the PCL instruction so that the inner-ring call can be retried.

flags.cond_fr — Is '0'b unless the frame is a condition frame (and is hence described by the structure CFH).

flags.cleanup_done — If '1'b, the nonlocal GOTO processor has cleaned up this frame by invoking its CLEANUP$ on-unit, if any, and resetting its sfh.ret_pb to point to a special code sequence to accomplish the unwinding of this stack frame. When '1'b, the former value of sfh.ret_pb can be found in sfh.tempsc(7:8) provided sfh.flags.efh_present is set.

flags.efh_present — If '1'b, the extension portion of this frame header has been validly initialized. This extension portion is marked EFH below. In the present implementation, this implies that at least one call to MKONU$ has been made, since MKONU$ is responsible for performing the initialization. If '0'b, members of this structure are not valid and can be used by the procedure for automatic storage.

flags.user_proc — If '1'b, this stack frame belongs to a nonsupport procedure; otherwise '0'b. If flags.user_proc is '1'b, sfh.owner_ptr is guaranteed to be valid and to point to an entry control block (ECB) that is followed by the name of the entrypoint.

flags.stk_cbits — If '1'b, then cond_bits exists within the stack frame header and should be used to determine whether to signal an exception condition. If '0'b, then flags.ecb_cbits is checked.

flags.lib_proc — If '1'b, then the procedure is a library routine.

flags.ecb_cbits — If '1'b, then ecb.cond_bits exists and should be used to determine whether to signal an exception condition. If both flags.stk_cbits and flags.ecb_cbits are '0'b, then flags.lib_proc is examined.

Note

If all three of the previous flag bits are reset ('0'b), then PL/I default condition handling is used.

flags.mbz — Is reserved and is '0'b.

flags.fault_fr — If '0'b, this frame was created by a regular procedure call; if '10'b, this frame is a fault frame (FFH) with valid saved registers; if '01'b, this frame is a fault frame (FFH) in which the registers have not yet been saved.

root.mbz — Is reserved and must be '0'b.

root.seg_no — Is the hardware-defined segment number of the stack root of the stack of which this frame is a member.

ret_pb — Points to the next instruction to be executed upon return from this procedure.

ret_sb — Contains the stack base belonging to the caller of this procedure, and hence also points to the immediate predecessor of this stack frame.

ret_lb — Contains the linkage base belonging to the caller of this procedure.

ret_keys — Contains the hardware-defined keys register belonging to the caller of this procedure.

after_pcl — Is a value pointing two halfwords beyond the procedure call (PCL) instruction that invoked this procedure.

hdr_reserved (EFH) — Is reserved for future expansion of the hardware-defined PCL stack frame header.

owner_ptr (EFH) — Points to the entry control block (ECB) of the procedure that owns this stack frame. This member must be initialized by the called procedure itself; the PCL instruction does not do it.

tempsc (EFH)	Is a fixed-position block of eight halfwords to be used as temporary storage by procedures called by this procedure that have a shortcall invocation sequence and hence have no stack frame of their own.
onunit_ptr (EFH)	Points to the start of a chain of on-unit descriptor blocks for this activation. If onunit_ptr is null, this activation has no on-unit blocks, except possibly for the condition CLEANUP$ as described below.
cleanup_onunit_ptr (EFH)	If nonnull, this activation has an on-unit for the special condition CLEANUP$, and cleanup_onunit_ptr points to the entry control block (ECB) for that on-unit procedure. It does not point to an on-unit descriptor block.
next_efh (EFH)	Points to the first on a chain of additional stack frame header blocks, so that these do not have to be allocated at the beginning of the stack frame. Presently, next_efh is always null.
reserved	Is reserved.
cond_bits	PL/I condition enable bits.

The entry control block (ECB) is described in the System Architecture Reference Guide.

The Standard Fault Frame Header (FFH)

Whenever a hardware fault occurs, the Fault Interceptor Module (FIM) is expected to push a stack frame with the standard format shown below. The standard fault frame header structure is:

```
dcl     1 ffh based,  /* standard fault frame header */
          2 flags,
            3 backup_inh bit(1),
            3 cond_fr bit(1),
            3 cleanup_done bit(1),
            3 efh_present bit(1),
            3 user_proc bit(1),
            3 mbz bit(9),
            3 fault_fr bit(2),
```

```
2 root,
  3 mbz bit(4),
  3 seg_no bit(12),
2 ret_pb ptr options (short),
2 ret_sb ptr options (short),
2 ret_lb ptr options (short),
2 ret_keys bit(16) aligned,
2 fault_type fixed bin,
2 fault_code fixed bin,
2 fault_addr ptr options (short),
2 hdr_reserved(7) fixed bin,
2 regs,
  3 save_mask bit(16) aligned,
  3 fac_1(2) fixed bin(31),
  3 fac_0(2) fixed bin(31),
  3 genr(0:7) fixed bin(31),
  3 xb_reg ptr options (short),
2 saved_cleanup_pb ptr options (short),
2 pad fixed bin;
```

flags.backup_inh	Is ignored by the condition mechanism for fault frames.
flags.cond_fr	Is '0'b in a fault frame.
flags.cleanup_done	Is set to '1'b by the procedure that unwinds the stack when it has cleaned up this fault frame. The old value of ffh.ret_pb has been placed in ffh.saved_cleanup_pb, provided flags.fault_fr is '10'b.
flags.efh_present	Is '0'b in a fault frame, implying that FIMs cannot make on-units.
flags.user_proc	Is always '0'b in a fault frame.
flags.mbz	Is reserved and is '0'b.
flags.fault_fr	Is '10'b if this frame is indeed a standard format FFH and the registers have been validly saved in ffh.regs; else is '01'b.
root.mbz	Is reserved and is always '0'b.
root.seg_no	Is the hardware-defined stack root segment number.

ret_pb	Points to the next instruction to be executed following a return from the fault. This is frequently also the instruction that caused the fault (the case for those faults defined by the System Architecture Reference Guide as backing up the program counter). If flags.cleanup_done is '1'b, ret_pb points to a special unwind code sequence, and its former value has been saved, if possible, in ffh.saved_cleanup_pb.
ret_sb	Contains the value of the SB register at the time of the fault, and hence usually points to the predecessor of this stack frame.
ret_lb	Contains the value of the LB register at the time of the fault.
ret_keys	Contains the value of the KEYS register at the time of the fault. This can be used to determine in what addressing mode the fault was taken.
fault_type	Is set by each FIM to the offset in the fault table corresponding to the fault that occurred (for example, a process fault results in a fault_type of '04'b3). This datum cannot be guaranteed valid, as it is not set indivisibly with the hardware-defined header information. Since FIMs usually set fault_type just after saving the registers, it is very unlikely for fault_type to be invalid.
fault_code	Is the hardware-defined fault code produced by the fault that was taken.
fault_addr	Is the hardware-defined fault address produced by the fault that was taken.
hdr_reserved	Is reserved for future expansion of the hardware-defined stack header.
regs	Is valid if flags.fault_fr is '10'b, and if valid, contains the saved machine registers at the time of the fault in the format produced by the RSAV instruction. For more information see the Instruction Sets Guide.
saved_cleanup_pb	Is valid only if flags.fault_fr is '10'b and flags.cleanup_done is '1'b, and if valid, contains the value that was in ret_pb before the latter was overwritten by the procedure that unwinds the stack.
pad	Exists only to make the size of this structure an even number of words.

The On-unit Descriptor Block

Each on-unit created by an activation is described to the condition mechanism by a descriptor block (except for the special condition CLEANUP$, which has no descriptor). These descriptor blocks are threaded together in a simple linked list, the head of which is pointed to by sfh.onunit_ptr. The format of an on-unit descriptor is:

```
dcl    1 onub based,  /* standard onunit block */
         2 ecb_ptr ptr options (short),
         2 next_ptr ptr options (short),
         2 flags,
           3 not_reverted bit(1),
           3 is_proc bit(1),
           3 specify bit(1),
           3 snap bit(1),
           3 mbz bit(12),
         2 pad fixed bin,
         2 cond_name_ptr ptr options (short),
         2 specifier ptr options (short);
```

ecb_ptr	Points to the entry control block (ECB) that represents the procedure or begin block to be invoked when this on-unit is selected for invocation.
next_ptr	Points to the next on-unit descriptor on the chain for this activation. A null pointer indicates the end of the list.
flags.not_reverted	Is '1'b if this on-unit is still valid and has not reverted; is '0'b if the on-unit has been reverted and is to be ignored by the condition-raising mechanism.
flags.is_proc	Is '1'b if this on-unit was made via a call to the primitive MKONU$; is '0'b if it was made via the PL/I ON statement.
flags.specify	Is '0'b if the condition name fully identifies which condition this on-unit block is to handle. Is '1'b if onub.specifier is a further qualifier for the condition.
flags.snap	Is '1'b if the snap option was specified in the PL/I ON statement that created this on-unit; '0'b otherwise.
flags.mbz	Is reserved and must be '0'b.
pad	Is reserved and must be 0.

cond_name_ptr	Is a pointer to a varying character string containing the condition name for which this on-unit is a handler. This name may be an incomplete specification if onub.flags.specify is '1'b.
specifier	Is valid only if onub.flags.specify is '1'b, and if valid, qualifies the condition name that is pointed to by onub.cond_name_ptr. The primary use of onub.specifier is for PL/I I/O conditions, in which the specification of the condition requires both a name and a file descriptor pointer.

8
Semaphores and Timers

REALTIME AND INTERUSER COMMUNICATION FACILITIES

PRIMOS supports user applications that have realtime requirements or that need to synchronize execution with other user programs.

The subroutine descriptions are divided into three parts. The first part describes routines that manipulate semaphores. The second part describes a routine used to signal the completion of some timed interval. The third part describes routines that cause a specific delay before resumption of processing.

SEMAPHORES

A set of subroutines provides access to Prime's semaphore primitives (wait and notify) and to internal timing facilities. The semaphore facility provides a means to coordinate multiple processes, providing that the processes involved all use the facility in the same way.

On time-sharing systems where more than one process can be active at the same time, there is often a need to coordinate the execution of multiple processes with one another. Such coordination is required when two or more processes cooperate to solve a common problem, or when multiple processes must use a common, limited resource.

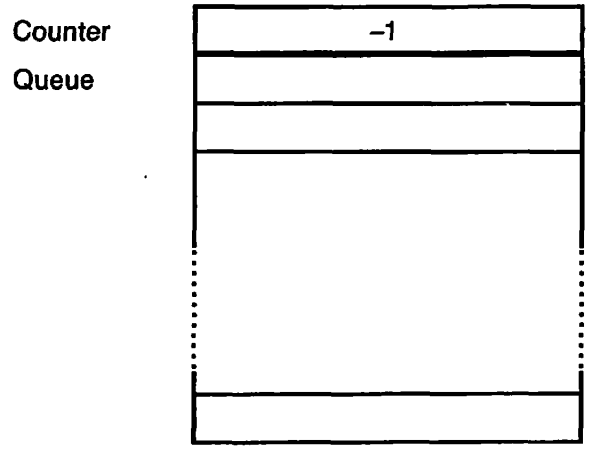
When multiple processes are working together as part of a larger system or to solve a common problem, it sometimes happens that one or more of the processes encounter a situation in which they cannot do any further work until some event, external to the process, happens. An example of this is a spooler that picks up print requests from a queue. When there are requests in the queue, the spooler services them. However, when the queue becomes empty, it can no longer do useful work and must wait for another process to give it something to do.

There are many resources on a time-sharing system that must be shared by all of the running processes. Included in the list are such things as devices that can have only one user at a time (such as a paper-tape punch), a section of code that performs a single operation, or files that are updated and read simultaneously by several programs.

The semaphore facility consists of some blocks of memory, which are called semaphores, and a set of software routines or hardware instructions that perform various operations on these blocks. There is no real connection between a semaphore and the event or resource with which it is associated. The use to which a semaphore is put is determined solely by the application programs that use it. All of the cooperating programs must agree on the meaning (or use) of a semaphore and use it the same way.

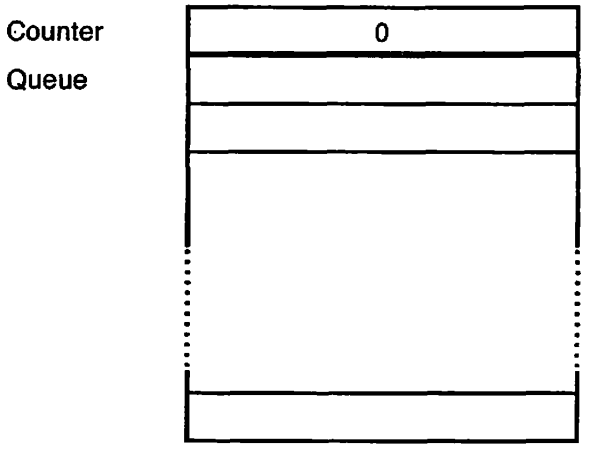
How a Semaphore Works

A semaphore consists of two parts: a counter and a queue (see Figure 8-1).



Resource Semaphore at Start
Figure 8-1

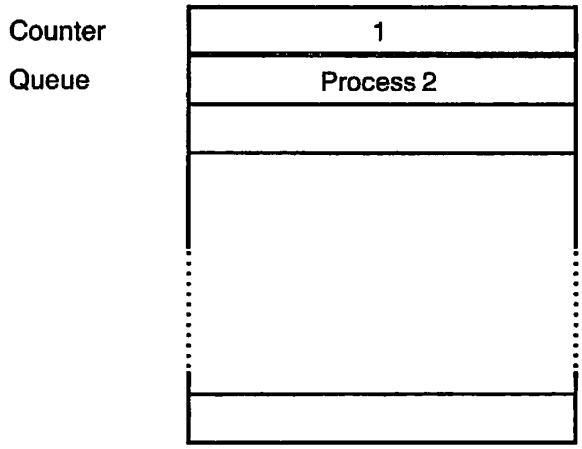
When a process wishes to wait for an event to happen or a resource to become available, it issues a wait call for the semaphore associated with that event or resource. The wait call will increment the counter for that semaphore and test its value. If the counter is less than or equal to 0, the process is allowed to proceed immediately and is not placed on the semaphore's queue (see Figure 8-2).



Resource Semaphore After Call by One Process
(Process 1 Is Using the Resource, No Processes Waiting)

Figure 8-2

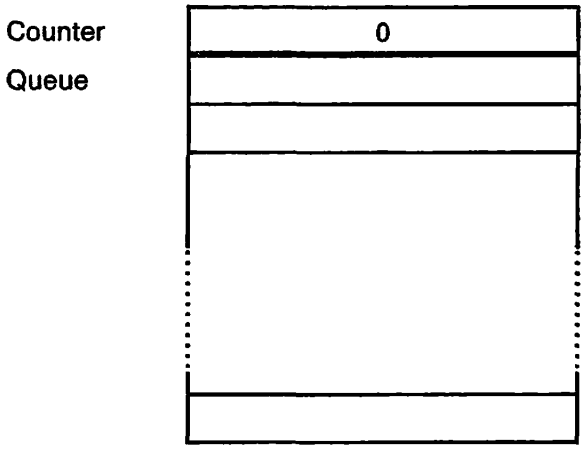
If, however, the counter is greater than or equal to 1 after being incremented, then the process is placed on the wait queue for the semaphore (see Figure 8-3). The process will not run again until it leaves this queue. Processes are placed on the queue in priority order with higher priority processes being placed closer to the head of the queue. Within a given priority, the processes are treated as a real queue -- first in, first out.



Resource Semaphore After Call by Second Process
(First Process Is Using the Resource)

Figure 8-3

When a process wishes to report that an awaited event has occurred, or that a resource has become available for use by other processes, it will call a notify routine for the semaphore associated with that event or resource (see Figure 8-4). The notify routine will first test the value of the counter for that semaphore. If the counter is greater than 0 (indicating that one or more processes are in the semaphore's queue), then the routine will remove one process from the top of the queue, thereby allowing that process to run again. Whether a process was dequeued or not, the routine will then decrement the counter by one.



Resource Semaphore After Notify by One Process
(Process 2 Is Now Using the Resource)

Figure 8-4

Normally, a semaphore's counter is preset to some value before the semaphore is used by any process. The value to which it is set depends on the nature of the software that will use the semaphore and on the purpose of the semaphore. Typical initial values are -1 and 0. A value of -1 allows the first process that waits on the semaphore to proceed immediately without being queued, as shown in Figures 8-1 through 8-4. This effect is desirable if the semaphore is used to coordinate the use of a shared resource. The resource is considered available until a process indicates its intent to use it. A value of 0 is appropriate for wait situations in which a process must wait until some condition exists or until an event occurs. The process that must wait for an event to happen does a wait operation on the semaphore, and is immediately put on the queue since the counter becomes greater than 0. When another process determines that the awaited event has occurred, it will notify the same semaphore, thus allowing the queued process to run.

When a process opens a named semaphore, and that process is the first to open that semaphore, then the SEM$OP routine will preset the semaphore's counter to a value of 0. If an initial value of -1 is required, then the process should notify the semaphore once after opening it. For named semaphores, SEM$OU also allows opening semaphores with initial values that are negative or 0. The minimum value is -32767. If the semaphore must be reset to its initial value of 0 at a later time, then a call can be made to the drain routine (see SEM$DR below).

Cooperation of Processes

It should be remembered that a semaphore is a structure that cooperating processes can use to control their access to resources, or to coordinate their execution. The operating system does not verify that the semaphore is being used correctly since the association between the semaphore and the event or resource is merely a convention adopted by the processes involved.

In order for the semaphore facility to work correctly, all processes that want to wait for an event or a resource must first wait on its associated semaphore before using the resource or assuming that the awaited event has occurred. There is nothing to stop the careless programmer from using a shared resource without first waiting on the appropriate semaphore. Such coding practices will most likely cause the entire subsystem of processes to malfunction.

PRIME SEMAPHORES

On Prime computers, a semaphore consists of two consecutive, nonpageable 16-bit halfwords of memory. The wait and notify operations are implemented in firmware and are usable by supervisor software only. So that users can use the semaphore facility, four calls have been created that perform the wait and notify operation on a set of semaphores that are reserved by the operating system for user programs:

- SEM$WT
- SEM$TW
- SEM$TN
- SEM$NF

There are 1024 named semaphores available to user processes, and 65 numbered semaphores.

Numbered Semaphores and Timers

Internal to PRIMOS is an array of 65 numbered semaphores reserved for the use of user processes. All reference to these semaphores is by the index of the semaphore, an integer from 0 to 64. Other than ensuring a valid semaphore number, PRIMOS makes no stipulations for semaphore use such as which users can access which semaphores, etc. Allocation and cooperative use of the semaphores is strictly under user control.

Of the 65 user semaphores, up to 15 can be used at any time as timed semaphores, that is, semaphores that are periodically notified by the system clock process. (See the SEM$TN routine.) Again, allocation of

timed semaphores is on a first-come/first-served basis, and nothing is done to prevent incorrect use of a timed semaphore.

Numbered semaphores are assigned by the operating system as wait or notify calls made to those numbers. No open or close request is necessary. It is your responsibility to use the number that has been agreed upon for a particular resource.

Named Semaphores

The operating system maintains a pool of semaphores that it can assign to user processes. When a process wishes to use one or more named semaphores, it must first ask the operating system to assign it to the process. The process requests access to named semaphores via an open routine. The user can request that multiple semaphores be assigned to it in a single call to this routine. The operating system returns a set of numbers to the process if it decides that the requested semaphores can be assigned to that process. The process uses these numbers in all subsequent calls to semaphore routines to indicate on which semaphore to perform the semaphore operation.

The operating system can tell when different processes wish to use the same set of semaphores by examining the parameters that they include in the call to the open routine.

See SEM$OP and SEM$OU below for more details on how to use the open call.

After a process has opened a set of semaphores, it can do any number of operations on those semaphores. The possible semaphore operations are given in the descriptions of the subroutines.

When a process has finished using the named semaphores that were assigned to it, it requests that the operating system close those semaphores, thus making them inaccessible to the process. When all processes finish using a given semaphore, then the operating system closes it and returns the memory space used by that semaphore to the operating system's free pool so that it may be assigned to other processes.

When a process logs out, all named semaphores that were opened by the process but not closed are closed automatically. If this process was the last user of a semaphore, the space used by the semaphore is returned to the free pool.

The routines that handle named semaphores are not available in R-mode.

CODING CONSIDERATIONS

Numbered vs. Named Semaphores

There are two methods by which a process can specify which semaphores it intends to use. Also, there are two sets of semaphores maintained by the operating system. One set is available to any process that wishes to use it, and its semaphores are identified by number. When a process wishes to use one of these semaphores, it specifies the number of the desired semaphore in the parameter list of the semaphore routines. This set of semaphores is called numbered semaphores. Numbered semaphores are easy to use, but they have a major drawback: there is nothing to prevent other processes from using the same semaphore for different purposes. Therefore, all users of the system must agree on the usage that each numbered semaphore will have; otherwise, confusion will result.

To eliminate the problems caused by the sharing of numbered semaphores, a second set of user semaphores was created. These are called named semaphores because they are associated with a file. Semaphores in this set cannot be used by a process until they are opened. Opening a semaphore means that the process must call the routine SEM$OP or SEM$OU, which will assign semaphores from the pool for the process to use. Each routine returns a set of numbers that can be used instead of numbered semaphore numbers in all other semaphore routine calls. Only valid semaphore numbers that have been assigned to a process by SEM$OP or SEM$OU can be used in subroutine calls that manipulate named semaphores. An attempt to use any other numbers will result in an error return from the routine.

To open a set of named semaphores, a routine must associate them with a file system object. SEM$OP will open a set of named semaphores and associate them with the name of a file in the current UFD of the process performing the open operation. SEM$OU will open a set of named semaphores and associate them with a file open on a particular file unit. In both cases, the process must have read access to the file.

Timers and Timeouts

When a process waits on a semaphore, it anticipates that it will be notified within a reasonable amount of time. If, for some reason, the process that is going to notify the semaphore fails to do so, all processes waiting on that semaphore will continue to wait, possibly for a very long time. To guard against processes waiting forever, a timer mechanism can be used.

Named Semaphore Timers: To prevent a process from waiting forever on a named semaphore, a special wait routine exists (called SEM$TW), which takes a semaphore number and a time value as parameters. The process waits on the specified semaphore until the semaphore is notified or until the specified amount of realtime has passed. The routine returns

a value to the process that indicates why the process was allowed to continue. A value of 0 means that the semaphore was removed from the wait queue because of a notify by another process. A value of 1 means that the process was allowed to continue because the specified time had elapsed without a notify on that semaphore. It is also possible for a value of 2 to be returned; this return value indicates that the process was stopped by someone pressing the BREAK key or CONTROL-P at the terminal controlling the process, and then typing START. This sequence causes the operating system to abort the process, thus removing it from the semaphore on which it was waiting, followed by a restart of the process at the wait call.

Numbered Semaphore Timers: The timer facility for numbered semaphores allows a semaphore to be automatically notified after a certain amount of time has passed. A user process tells the operating system, via a subroutine call, that a timer is to be associated with a numbered semaphore. The process also specifies the amount of time that should pass before the operating system notifies the semaphore. When this amount of time has passed, the operating system notifies the semaphore.

Much care is needed when coding programs that use semaphores with this kind of timer. If another method is not used besides the semaphore to indicate that the awaited event has actually occurred, then a notify caused by a timer cannot be distinguished from a notify caused by a process. The processes using the semaphore should, therefore, be coded so that they can verify that a notify by another process has occurred before using the resource protected by the semaphore. The action that is taken when a timer notifies the semaphore should be agreed upon by all of the processes using the timed semaphore.

PITFALLS AND HOW TO AVOID THEM

External Notifies

When a semaphore is notified for some reason other than an explicit call to the notify routine, that notify is called an external notify; that is, it originated from a source external to the processes that are using the semaphore. Some of the reasons that an external notify may occur are listed here.

Expiration of a Timer: When a timer is set for a numbered semaphore, and that timer expires, the operating system will notify the semaphore. This semaphore will look like an external notify to the processes that use the semaphore; the fact that the notify is external can be detected if the processes are coded properly. (See Coding Suggestion below.)

The notify caused by a timeout can be useful in cases when the process that is supposed to notify the semaphore is prone to being aborted.

The notify initiated by the operating system prevents processes from waiting forever.

Use of timers with named semaphores causes a code to be returned to the process that indicates when a timeout has occurred.

Malfunctioning Process: Like all other programs, processes that are supposed to be using a semaphore sometimes do not behave properly. Malfunctioning programs can do extra notify calls and thereby cause what appear to be external notifies. Also, processes that are not supposed to be using a numbered semaphore may decide to use it anyway. Unless the semaphore can be protected from such interference, then what appears to be an external notify will result.

Process Quit: The semaphores that a user process can access on a Prime system are called quittable semaphores. This means that a process that is waiting on a semaphore can be stopped by pressing the BREAK key or CONTROL-P at the terminal controlling the process. When a process is stopped by this means, and then continued (by using the PRIMOS START command), the process will reexecute the wait operation.

Coding Suggestion: Since semaphores can be notified by breaks and timeouts as well as by explicit calls to SEM$NF, and since this could cause malfunctions in a subsystem, it is always best to code in such a way that this situation can be detected. This means that a process should not rely solely on the semaphore to indicate that a resource is really available or that an event has actually occurred. A good practice is to have one additional method, besides the semaphore, to indicate what the current state of the resource or event is.

One such method is to have a halfword in shared memory (accessible by all cooperating processes) that is set to indicate that the event has really occurred or that a resource is free. Before a process notifies a semaphore, it sets this halfword to an agreed value. When the process is allowed to proceed from a semaphore wait, it should check the value contained in that halfword. If the halfword contains the value, it will know that the semaphore was notified by a cooperating process, and not by the operating system. In this case, the process will clear the halfword, do its processing, and reset the halfword to the agreed-upon value just before notifying the semaphore. If a process proceeds from a wait call and the halfword is not set to the agreed-upon value, it can assume that the operating system notified the semaphore and can re-issue the wait call.

Infinite Waits

It is possible to create a situation in which one or more processes are waiting on a semaphore, and there are no processes running that will ever notify that semaphore. The following are methods of creating this situation.

Multiple Waits: If a process issues a wait call, and is not queued, and then continues to re-issue the wait call without intervening notifies, that process will eventually cause the semaphore count to become greater than 0 and the process will wait. This of course assumes that there is not another process somewhere doing multiple notifies.

In the case of a resource-protection semaphore, if all other processes obey the rules, they will wait on this semaphore before they notify it. They will therefore queue up behind the multiple-waiter process. Eventually, all the processes of the subsystem will become queued on the semaphore queue, and no process will remain to notify the semaphore.

Aborted Notifiers: Another way of causing infinite waits is to abort a process that would, under normal circumstances, notify a semaphore. If this is the only process that will do notifies on the semaphore, then all other processes that wait on it will wait forever.

Coding Suggestion: Infinite waits can be avoided by associating a timer with the semaphore. This will guarantee that one or more processes will eventually be removed from the wait queue. Extra coding must be done in the processes, however, so that a timeout can be identified as such, and so that appropriate action can be taken. This code should determine whether the process that should have notified the semaphore is still running or not. If it is running, the notify is considered external and the process re-issues the wait call. If the potential notifiers have all been aborted, appropriate recovery action should be initiated.

Deadly Embrace

When multiple semaphores are being used, a situation called deadly embrace can occur. This happens when two processes gain rights to use a resource by waiting on the appropriate semaphore for that resource, and then each attempts to acquire the resource that is being used by the other process. Neither process will ever notify the semaphore for the resource it holds (it is waiting to get access to a second resource), and no other process will ever notify the semaphores (since each resource is held already by one of the two processes). Therefore, both processes will wait forever.

This situation can neither be detected nor prevented by the semaphore facility. It can be prevented, however, by the processes using the semaphores, if the following procedure is used.

Each semaphore that a system of processes will use is assigned a different number; this number will be called the semaphore's level number. Processes can only issue a wait call for a semaphore whose level number is greater than the level number of any semaphore it has waited on but has not yet notified. For example, if the level numbers for three semaphores are 1, 2, and 3, and a process has waited on the second semaphore (level 2), but has not yet notified it, then the process can legally issue a wait for the third semaphore (level 3) but not for the first, since level 1 is numerically less than level 2.

This technique, if strictly followed, makes deadly embrace situations impossible. It is sometimes practical for processes to call a routine that checks for level number violations before issuing a wait call. If all processes use this routine instead of the wait routine, then deadly embrace is prevented.

LOCKS

Locks, like semaphores, are a method that programs or processes can use to coordinate their usage of some resource. Before a process attempts to use a resource that is protected by a lock, it calls a routine that grants or denies permission to use the resource or causes the process to wait until the resource becomes free. When the process has been given permission to use the resource, it is said to hold the lock on that resource. When the process is through using the resource, it calls another routine to indicate that it is done. This operation is called giving up the lock, or releasing the lock, on that resource.

Various types of locks exist, some of which will be discussed in this section.

Some types of locks behave very much like semaphores and, in fact, many types of locks can be coded with the use of semaphores. Semaphores, unlike locks, allow a small, well-defined set of operations to be performed, but the uses and types of locks that can be coded vary greatly.

Mutual Exclusion

Mutual-exclusion locks are used when only one or a few processes are allowed to use a resource at any given time. When a process requests ownership of a lock for the resource, it is given the lock if no other process currently holds it. If the lock is held by another process, all others must wait until the one holding the lock gives it up.

This type of lock can be implemented directly with the use of semaphores. Requesting the lock is equivalent to a wait operation on a semaphore; giving up the lock is equivalent to a notify of that semaphore.

Since external notifies may occur, it is a good practice to expect them and to code in such a way that they can be detected and ignored.

N1 Locks

N1 locks are used to protect objects that can be read and modified simultaneously, such as files and data bases. This type of lock allows any number of users to read the object, or one process to modify the object. In the PRIMOS filing system, this is referred to as an N readers or one writer lock. When a process requests permission to read the object, such permission is granted immediately, as long as there is not currently a process modifying it. Requests to gain access to the object for modification are granted only if there are no other readers or writers using the object. If another process is using the protected object, the writer is placed on a queue and must wait until all current users of the resource indicate that they are done. If a writer is waiting to use the resource, then no other requests for use of the object are granted until that process has used the object. This prevents readers from gaining access to the object and causing the writer's request to be delayed indefinitely.

When a writer is given access to the object, all other requests for access are queued. When the writer finishes, the other requests are processed.

Use of an N1 lock on a file eliminates data loss that can sometimes occur when multiple processes are allowed to update the same file simultaneously.

Producers and Consumers

In many computer systems, certain processes create work that must be processed, such as device drivers that read data from a device that must be routed to the correct place, or print programs that place data files into spool queues to be printed. These work-producing processes are called producers.

Other processes in a system process the work created by the producers. These processes are called consumers. Examples of consumers include a user process that manipulates data coming into the system from a peripheral device, or a spooler that prints files in response to a user's print requests.

The coordination required between producer processes and their corresponding consumer processes can be achieved with the use of producer-consumer locks.

Producers call a routine that indicates that there is work to process. The routine keeps track of the number of producers that have called it; each call indicates that another unit of work is available. Consumers, on the other hand, call a routine that checks to see if there is any work to do. If there is no work, the routine causes the consumer process to wait until there is work, that is, a producer calls the I-have-work-to-do routine. If there is work to do, the consumer process is allowed to continue, and the counter of units of work left to do is decremented.

This lock can be coded directly with semaphores. A semaphore, with its counter initialized to 0, serves as the locking mechanism. Producers notify the semaphore, causing it to become negative; consumers wait on the semaphore, causing it to rise toward 0. If there is no work to do (semaphore counter equal to 0), then a consumer will be queued, when it waits on the semaphore, until work becomes available.

Note that there can be any number of producers or consumers. If multiple consumers wait for work, and there is none to do, then the semaphore counter will contain a value equal to the number of queued consumer processes. A notify by a producer allows one of the consumers to proceed.

Since semaphores are subject to external notifies, it is advisable that a counter, other than the counter that is a part of the semaphore, be maintained to indicate how much work is available for consumer processes. Producers increment this counter; consumers take work from the work queue and decrement this counter. If a consumer is notified out of the semaphore queue and the counter does not match the semaphore counter, then it can assume that an external notify has occurred.

Record Locks

When many processes must update a file, and speed is important, it is not practical to use a lock that protects the entire file, since any update request would lock all other processes out of the file. Considerable overlap in processing can usually be achieved if just the portion of the file that is being updated by a process is locked. Usual units to lock are the record or the page being updated.

If the file is large, then it becomes impractical or impossible to have an individual lock for each record or page to be protected. One way of overcoming this difficulty is to assign locks from a pool on a temporary basis. When a process wishes to update a record, for example, it requests a lock by passing the record number in question to the lock routine. If there is currently no one holding a lock on that record (the lock routine scans its list of locks being held by other processes), then a lock is assigned from a free pool and the record

number supplied is remembered. If a lock is requested for a record that is currently locked by another process, then the second and subsequent requesters of the lock are forced to wait. When the last holder of a lock gives up the lock, and there are no other processes waiting to use the record protected by that lock, then the lock itself is returned to the pool of free locks. It can then be used for other record locks.

In general, the pool of locks needs to be as large as the expected maximum number of records that can be locked at any given time. It is the lock routine's responsibility to manage the lock pool and to deal with the problems that arise when there are no more free locks in the pool. One method of dealing with this situation is to use a no-free-locks semaphore. If there are no free locks in the pool, the process requesting the lock is forced to wait on this semaphore. The lock routine notifies this semaphore when a lock becomes available.

Notice that record locks are really mutual-exclusion locks; however, the object that is being protected by any given lock changes with time. The lock routine must include a small data base that is used to remember what is being protected by each lock.

SEMAPHORE ROUTINES

This section describes the following subroutines:

Routine	Function
SEM$CL	Releases (closes) a named semaphore.
SEM$DR	Drains a semaphore.
SEM$NF	Notifies a semaphore.
SEM$OP	Opens a set of named semaphores.
SEM$OU	Opens a set of named semaphores.
SEM$TN	Periodically notifies a semaphore.
SEM$TS	Returns number of processes waiting on a semaphore.
SEM$TW	Waits on a specified named semaphore, with timeout.
SEM$WT	Waits on a semaphore.

SEM$CL

Purpose

SEM$CL releases (closes) a semaphore.

Usage

```
DCL SEM$CL ENTRY (FIXED BIN, FIXED BIN);

CALL SEM$CL (snbr, code);
```

Parameters

snbr

INPUT. A semaphore number; it must be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr.

Discussion

When a process no longer needs a named semaphore, it can tell the operating system that it is done with the semaphore by calling SEM$CL. This call closes the semaphore. After this call, the specified semaphore number cannot be used again by the process unless that same number is reassigned by another call to SEM$OP or SEM$OU.

When a process logs out, all semaphores that were opened by that process but not explicitly closed are automatically closed by the operating system.

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.

SEM$DR

Purpose

SEM$DR resets ("drains") the specified semaphore counter to 0.

Usage

```
DCL SEM$DR ENTRY (FIXED BIN, FIXED BIN);

CALL SEM$DR (snbr, code);
```

Parameters

snbr

INPUT. A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or it can be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr.
E$BDAT	Indicates bad data supplied; the System Administrator should be notified.

Discussion

The counter is set to 0 if, at the time of the SEM$DR call, the semaphore's counter is less than or equal to 0. If, however, the counter is greater than 0, then notifies are done on the semaphore until the counter reaches 0. This causes all processes that were waiting on the semaphore to be removed from the wait queue of the semaphore.

It is possible for processes to be placed on the wait queue while this call is executing. These added processes may not be removed when the SEM$TS call returns to its caller.

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.

SEM$NF

Purpose

SEM$NF releases the next process waiting on a semaphore ("notifies" the semaphore).

Usage

```
DCL SEM$NF ENTRY (FIXED BIN, FIXED BIN);

CALL SEM$NF (snbr, code);
```

Parameters

snbr

INPUT. A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or it can be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine (FIXED BIN).

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	Indicates that an invalid value was supplied for snbr.
E$SEMO	Indicates that the semaphore count became too small to be decremented.
E$BDAT	Indicates that bad data was supplied; the System Administrator should be notified.

Discussion

The notify and wait operations are the basic functions that can be performed on a semaphore. A notify decrements the semaphore's counter and releases the first process from the wait queue, if there are any processes waiting.

A wait increments the semaphore's counter and places the process on the semaphore's queue if the counter becomes greater than 0. Processes are queued first-in/first-out within process priority; higher priority processes are queued before those with lower priority.

The wait procedure is SEM$WT. This is described later 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.

SEM$OP

SEM$OU

Purpose

These routines open a semaphore.

Usage

```
DCL SEM$OP  (CHAR(32),  FIXED BIN, FIXED BIN, (*)FIXED BIN, FIXED BIN);

CALL SEM$OP (fname, namlen, snbr, ids, code);

DCL SEM$OU (FIXED BIN, FIXED BIN, (*)FIXED BIN, FIXED BIN, FIXED  BIN);

CALL SEM$OU (funit, snbr, ids, init_val, code);
```

Parameters

fname

INPUT. A filename, as discussed below.

funit

INPUT. The number of a file unit that has already been opened.

namlen

INPUT. The number of characters in fname.

snbr

INPUT. A number that specifies how many semaphores are to be opened by this call.

ids

OUTPUT. An array of semaphore numbers; one number is returned for each semaphore that was successfully opened. There must be at least snbr elements in ids.

init_val

INPUT. The initial value (-32767 to 0) to be assigned to the semaphore.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr, namlen, or init_val.
E$IREM	A file that is on a remote disk was specified in the fname parameter -- remote files cannot be used as parameters to this call.
E$FUIU	Either the user has all available file units opened, or there are no available named semaphores.
E$UNOP	Unopened file unit.
E$BUNT	Bad file unit.

It is also possible that code will be set to any error code that can be returned by the SRCH$$ routine.

Discussion

To open a set of named semaphores, a call must associate them with a file system object. SEM$OP opens a set of named semaphores associated with the name of a file in the current UFD of the process performing the open operation. If the process has at least read-access rights to the file, it will be assigned the semaphores. Each semaphore is initialized to 0. SEM$OU opens a set of named semaphores, associating with them a file open on a particular file unit. As before, if the process has at least read-access rights to the file, it is assigned the semaphores. Unlike SEMOP, SEMOU allows each semaphore within the set to be initialized to a nonpositive value, not less than -32767 decimal. All calls to either SEM$OP or SEM$OU that use the same file result in the same semaphore numbers being returned.

It is possible for a number of processes to have access to a set of semaphores while other processes are denied access to the same semaphores. These semaphores are called protected or named semaphores and are discussed in the introduction to this chapter.

To access a named semaphore, a call must be made to SEM$OP, which grants or denies access to the semaphore. The process supplies a filename to the call. If the specified file can be accessed for read access, subject to file system and ACL protections, then the user is given access to the requested semaphores. Multiple semaphores can be opened in a single call by supplying the number of semaphores needed in the snbr parameter.

If access is granted to the semaphores, then the call returns an array of semaphore numbers in the ids parameter. One number is returned for each semaphore requested in snbr, assuming enough semaphores exist in the system pool. A semaphore number of 0 is returned if a semaphore could not be assigned. In addition, code is nonzero if one or more semaphore numbers could not be assigned. The values returned in ids should be examined to determine which semaphores were opened (nonzero value returned), and which were not (0 value returned).

The semaphore numbers returned should be used in all other semaphore calls as the semaphore number parameter. SEM$OP takes a filename and returns semaphore numbers; SEM$OU takes a file unit; the rest of the calls accept only a semaphore number.

If different processes call SEM$OP or SEM$OU and specify the same filename or file unit, the same semaphore numbers will be returned to each process. This allows multiple processes of a subsystem to reference common semaphores.

If a call to the open routine specifies the same filename or unit number as a previous call to open, and a larger number of semaphores is requested, then new semaphores are acquired from the system pool to make up the difference between the number currently open (with that filename or unit number) and the number requested in the call. Other processes cannot use these newly assigned semaphores unless they explicitly open them via a call to the open routine.

When the first process opens a named semaphore, the operating system sets the value of the semaphore counter to 0 or to the number specified by SEM$OU. Subsequent opens of the semaphore do not alter the value of the counter. If a process opens the same semaphores more than once, then the same semaphore numbers are returned for each call. No matter how many times a process opens a semaphore, it need only close that semaphore once. This removes the burden of counting to be sure that equal numbers of open and close calls are done.

Named semaphores can only be opened for files that reside on a local computer system. Attempts to open named semaphores with filenames that are on remote disks will result in failure; no semaphore numbers are assigned and code are set to E$IREM.

If a file that was used in a call to SEM$OP or SEM$OU is deleted or renamed while the semaphores assigned by such a call are still open, or if the disk on which that file resides is shut down, then the open semaphores will continue to be accessible to the processes that already have them open. New processes will not be given access to those

semaphores, even if the disk is added again, or if a file is created with the same name as the one that was renamed or deleted. Processes that have the semaphores open can continue to use them until they are closed via a call to SEM$CL.

If a process logs out before all named semaphores have been closed, then those that are still open will be automatically closed by the operating system.

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.

SEM$TN

Purpose

This operation causes the operating system to notify the specified semaphore on a periodic basis. This procedure can be used only on numbered semaphores.

Usage

```
DCL SEM$TN ENTRY (FIXED BIN, FIXED BIN(31), FIXED BIN(31), FIXED BIN);

CALL SEM$TN (snbr, int1, int2, code);
```

Parameters

snbr

INPUT. A semaphore number; it must be a number in the allowable range for numbered semaphores (0-64).

int1

INPUT. The amount of clock time (in milliseconds) that will pass before the system notifies the semaphore the first time.

int2

INPUT. The amount of clock time (in milliseconds) that will pass before the semaphore is notified the second and subsequent times. If int2 is 0, then the semaphore will only be notified once: after int1 milliseconds. Specifying both int1 and int2 as 0 will remove a previous timer request from the semaphore. This is necessary when a previous SEM$TN call was made with int1 and int2 both nonzero.

If a call is made to SEM$TN that specifies a semaphore that already has an active timer request, then the values of int1 and int2 specified in the latter call overwrite the values stored in the active timer.

Note

It is possible to indefinitely delay a notify caused by a timeout by making repeated calls to SEM$TN.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr, int1, or int2.
E$NTIM	The operating system has no more timers available.
E$BDAT	Bad data supplied; the System Administrator should be notified.

Discussion

The operating system maintains a limited number of timers for numbered semaphores. Currently, there are a total of fifteen such timers per system.

The time intervals, quoted in milliseconds, are truncated to the nearest tenth of a second before being 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.

SEM$TS

Purpose

SEM$TS tests the counter for the number of processes waiting in the queue for a semaphore.

Usage

```
DCL SEM$TS ENTRY (FIXED BIN, FIXED BIN) RETURNS (FIXED BIN);

sval = SEM$TS (snbr, code);
```

Parameters

sval

RETURNED VALUE. The current value of the specified semaphore's counter halfword.

snbr

INPUT. A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or a number assigned to a named semaphore by the SEM$OP or SEM$OU routine.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr.

Discussion

This operation returns in sval the current value of the counter, for the semaphore numbered snbr.

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.

SEM$TW

Purpose

This routine allows a process to wait on the specified semaphore until it is taken off the wait queue by a notify, or until a specified amount of realtime has elapsed, whichever comes first. It is used only for named semaphores.

Usage

```
DCL SEM$TW ENTRY (FIXED BIN, FIXED BIN, FIXED BIN);

CALL SEM$TW (snbr, int1, code);
```

Parameters

snbr

INPUT. A semaphore number; it must be a number assigned to a named semaphore by the SEM$OP or SEM$OU routine.

int1

INPUT. A time interval expressed in tenths of a second of clock time.

code

OUTPUT. A value that indicates why the process was allowed to continue, or a standard error code. Possible values are

0	The process was notified by a call to SEM$NF.
1	The specified amount of time has elapsed and the process has not yet been notified out of the wait queue.
2	The process was aborted, for example, by a quit or forced logout.
E$BPAR	An invalid value was supplied for snbr or int1.
E$BDAT	Bad data supplied; the System Administrator should be notified.

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.

SEM$WT

Purpose

SEM$WT places a process in the queue for a semaphore.

Usage

```
DCL SEM$WT ENTRY (FIXED BIN, FIXED BIN);

CALL SEM$WT (snbr, code);
```

Parameters

snbr

INPUT. A semaphore number; it can be either a number in the allowable range for numbered semaphores (0-64), or a number assigned to a named semaphore by the SEM$OP or SEM$OU routine.

code

OUTPUT. Standard error code. Possible values are

0	Success.
E$BPAR	An invalid value was supplied for snbr.
E$BDAT	Bad data supplied; the System Administrator should be notified.

Discussion

The notify and wait operations are the basic functions that can be performed on a semaphore. Notify decrements the semaphore's counter and releases the first process from the wait queue, if there are any processes waiting.

Wait increments the semaphore's counter and places the process on the semaphore's queue if the counter becomes greater than 0. Processes are queued first-in/first-out within process priority; higher priority processes are queued before those with lower priority.

The notify procedure is SEM$NF, described later 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.

LIMIT TIMER ROUTINE

This section describes the following subroutine:

Routine	Function
LIMIT$	Sets and reads various timers.

LIMIT$

Purpose

LIMIT$ allows the setting of various timers within PRIMOS, each generating a signal if expired. The timer values may also be read.

Usage

```
DCL LIMIT$ ENTRY (FIXED BIN, FIXED BIN(31), FIXED BIN, FIXED BIN);

CALL LIMIT$ (key, limit, res, code);
```

Parameters

key

INPUT. This key is split into two 8-bit functions. The right half is as follows:

1 Read the limit.

2 Set the limit.

The left half is as follows:

1 CPU limit in seconds.

2 Login limit in minutes.

5 CPU watchdog in seconds.

6 Realtime watchdog in minutes.

7 Realtime watchdog in seconds.

limit

INPUT. The time to be set in minutes or seconds.

res

INPUT. Reserved--must be zero.

code

OUTPUT. Standard error code. Possible values are

0	No error.
E$BKEY	Invalid value specified in key.
E$BPAR	Invalid value specified for limit, or nonzero value specified for res.

Discussion

A watchdog timer is a timer that starts counting when LIMIT$ is called, and "expires" when the indicated realtime or CPU processing time has elapsed.

If LIMIT$ is called to set the CPU or realtime (login) limit, the user will be logged out when the interval expires. If LIMIT$ is called to set a realtime watchdog, then the condition ALARM$ will be signalled when the interval expires. If LIMIT$ is called to set a CPU watchdog, then the condition CPU_TIMER$ will be signalled when the interval expires.

Any of the timers can be canceled by calling LIMIT$ with the key value that was used for setting the timer and a limit value of 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.

PROCESS DELAY ROUTINES

This section describes the following subroutines:

Routine	Function
SLEEP$	Suspends a process for a specified interval.
SLEP$I	Suspends a process (interruptible).

SLEEP$

Purpose

SLEEP$ suspends a process for a specified interval.

Usage

```
DCL SLEEP$ ENTRY (FIXED BIN(31));

CALL SLEEP$ (interval);
```

Parameters

interval

INPUT. A variable containing the interval, in milliseconds, for which execution is to be suspended.

Discussion

Execution of the user process is suspended for the specified interval. An interval less than 0 will have no effect. A QUIT and START from the user terminal will cause immediate reexecution of the SLEEP$ call.

Note

Although the sleep interval is specified in milliseconds, SLEEP$ truncates it to an accuracy of tenths of seconds.

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.

SLEP$I

Purpose

This procedure suspends the process for a specified interval.

Usage

```
DCL SLEP$I ENTRY (FIXED BIN(31));

CALL SLEP$I (interval);
```

Parameters

interval

INPUT/OUTPUT. Defines the delay interval in units of tenths of a second. The user's variable is continually updated with the amount of time remaining.

Discussion

Execution of the user process is suspended for interval tenths of a second. An interval less than 0 will have no effect. If the wait is interrupted (for example, by a terminal QUIT), an on-unit can read the value of the parameter to determine the amount of time remaining to sleep. This contrasts with SLEEP$, which does not update its parameter.

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.

9
Message Facility

The PRIMOS MESSAGE feature includes calls for sending and receiving interuser messages. The subroutines can also set and query a user's willingness to receive messages. Messages may be sent in either immediate mode or deferred mode (to be delivered at command level only), and may be addressed with either a user name or a user number. Reception may also be controlled, allowing users to select one of three modes of reception: receive at any time, receive at command level only, or never receive.

MESSAGE FACILITY ROUTINES

This section describes the following subroutines:

Routine	Function
MSG$ST	Returns the receiving state of a user.
MGSET$	Sets the receiving state for messages.
RMSGD$	Receives a deferred message.
SMSG$	Sends an interuser message.

MSG$ST

Purpose

MSG$ST allows the caller to determine the receive state of processes. If the caller supplies a specific user number, the receive state and user name of that process are returned. If the caller supplies a user name, the user number and receive state of the most permissive user with the specified name are returned.

Usage

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

CALL MSG$ST (key, user_num, system_name, system_name_len,
             user_name, user_name_len, receive_state);
```

Parameters

key

INPUT. Can be either of the following:

K$READ	Return the user's name and state for user user_num on system system_name.
2	Return the user's number and state for user user_name on system system_name.

user_num

INPUT or OUTPUT. The user number of the process. If key = K$READ, user_num is provided by the user. If key = 2, user_num is provided by the user and updated by the subroutine.

system_name

INPUT. The name of the system on which the desired process is found.

system_name_len

INPUT. The length of system_name in characters. If system_name_len = 0, the local system is assumed.

user_name

INPUT or OUTPUT. The user name of the process. If key = K$READ, this parameter is returned by the subroutine. If key = 2, this parameter is provided by the user.

user_name_len

INPUT. The length of user_name in characters.

receive_state

OUTPUT. The receive state of the process. This parameter can be any of the following:

K$ACPT	Accepting all messages.
K$DEFR	Accepting deferred messages only.
K$RJCT	Rejecting all messages.
K$NONE	User does not exist.
K$BKEY	Invalid state, bad key.
K$BREM	Invalid state, bad system_name.

Discussion

If key = 2, the search for the users whose names match user_name begins at user_num + 1. This will help the programmer to scan all users with the same name, by setting user_num to 1 at the first call and repeating the call until a nonzero code is returned.

Loading and Linking Information

V-mode and I-mode: No special action.

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

R-mode: Not available.

MGSET$

Purpose

MGSET$ is used to set the message receive state of the calling process. The receive state determines the willingness of the process to accept messages sent to it.

Usage

```
DCL MGSET$ ENTRY (FIXED BIN, FIXED BIN);

CALL MGSET$ (key, code);
```

Parameters

key

INPUT. A standard system key that specifies the receive state to be set.

K$ACPT	Accept all messages.
K$DEFR	Accept only deferred messages.
K$RJCT	Reject all messages.

code

OUTPUT. Standard error code.

E$BKEY	Bad key.
0	No error.

Discussion

There are three possible states that a process may have: accept all messages, accept only deferred messages, and reject all messages. Messages that are deferred are not necessarily delivered immediately when sent, but rather are stored in buffers by the system and delivered later. Deferring messages allows the receiver to accept messages at convenient times rather than at times convenient to the sender. Users may explicitly request waiting deferred messages via the RMSGD$ call, or they may allow the system to deliver deferred messages automatically after PRIMOS commands complete their execution.

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.

RMSGD$

Purpose

RMSGD$ returns waiting deferred messages to the caller. This routine does not return immediate messages. Users wishing to obtain all messages via this routine must inhibit immediate messages by setting their receive state to receive only deferred messages (see MGSET$ with a key of K$DEFR).

Usage

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

CALL RMSGD$ (from_name, from_name_len, from_num, system_name,
             system_name_len, time_sent, text, text_len);
```

Parameters

from_name

OUTPUT. The user name of the sender.

from_name_len

INPUT. The maximum length of from_name in characters.

from_num

OUTPUT. The sender's user number.

system_name

OUTPUT. The name of the system from which the message was sent.

system_name_len

INPUT. The maximum length of system_name in characters.

time_sent

OUTPUT. The time, in minutes past midnight, at which the message was sent. If no message is returned, time_sent is set to -1.

text

OUTPUT. The text of the message.

text_len

INPUT. The maximum length of text.

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.

SMSG$

Purpose

SMSG$ sends a message. Messages may either be sent immediately or deferred.

Usage

```
DCL SMSG$ ENTRY (FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN, CHAR(*),
                 FIXED BIN, CHAR(*), FIXED BIN, (258) FIXED BIN);

CALL SMSG$ (key, to_name, to_name_len, to_user_num,
            to_system_name, to_system_len, text, text_len,
            error_vector);
```

Parameters

key

INPUT. Specifies the type of message, immediate or deferred.

0	Deferred message. Messages are buffered and delivered at the receiver's convenience.
1	Immediate message. Messages are delivered immediately when sent.

to_name

INPUT. The user name of the user to whom the message is to be sent. If to_name is nonblank, the message is sent to all users logged in under that name. If to_name is blank, the message is sent by to_user_num, and to_name is ignored.

to_name_len

INPUT. The length of to_name in characters.

to_user_num

INPUT. The user number of the user to whom the message is sent. If to_user_num is positive, to_name is ignored. If to_user_num is zero and to_name is blank, the message is sent to the operator.

to_system_name

INPUT. The name of the node to which the message is to be sent. If the target system is local (indicated by to_system_len being zero), to_system_name is ignored.

to_system_len

INPUT. The length of to_system_name in characters. If to_system_len is zero, the local system is assumed.

text

INPUT. The text of the message. Messages may be up to 80 characters in length, and either blank-padded or terminated with a newline. Only printable characters and the bell character are printed by the operating system.

text_len

INPUT. The length of text in characters.

error_vector

INPUT/OUTPUT. An array that reports the success or failure of the call. Its size can range from 4 through 258. Its elements have the following meanings:

error_vector(1) The standard error code status returned by the subroutine.

E$NRCV	Operation aborted because send does not have receive enabled.
E$UADR	Unknown addressee.
E$UDEF	Receiver not receiving.
E$PRTL	Operation partially blocked.
E$NSUC	Operation failed.
0	Operation succeeded.

error_vector(2) Three less than the total number of elements in error_vector. Normally set to the number of configured users (256). Provided by the user.

Note

This is both an input and output parameter. On input, if error_vector(2) is set to less than the number of users configured (KUSR), only that many elements will be set from error_vector(3) on. If error_vector(2) is greater than KUSR, it will be set to KUSR. Thus, if you are not interested in the information, this large buffer need not be reserved.

error_vector(3) An overall network error code/returned by the subroutine.

XS$CLR	Connect cleared.
XS$BPM	Unknown node address.
XS$DWN	Node not responding.

error_vector(4-258) An optional status vector whose length is the value of error_vector (2). If supplied, each element is a status code returned by the subroutine, indicating success or failure to send a message to user number n - 3, where n is the index into error_vector. For example, error_vector (10) is the status for user number 7.

E$UBSY	User busy, please wait.
E$UNRV	User not receiving now.

Discussion

Immediate messages are delivered to the recipient at the time the message is sent. Deferred messages are held in a system buffer until the receiver requests them. (Deferred messages are also delivered to a user automatically after each PRIMOS command completes execution.) Messages may be sent to other processes by addressing them to either their user numbers or their user names. If user name is used, all users with that name will receive the message.

A noninteractive (phantom or batch) process does not have messages delivered at command level. Consequently the "immediate" option is not available. The process can receive messages using RMSGD$.

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.

10
Superseded Routines

This chapter lists routines considered obsolete or superseded, which Prime continues to support.

SUPERSEDED ROUTINES

This section describes the following subroutines:

Routine	Function
DISPLY	Updates sense light settings.
ERRSET	Sets ERRVEC (a system error vector).
GETERR	Returns ERRVEC contents.
OVERFL	Checks if an overflow condition has occurred.
PHANT$	Starts a phantom process.
PRERR	Prints an error message.
SLITE	Sets the sense light on or off.
SLITET	Tests sense light settings.
SSWTCH	Tests sense switch settings.
TEXTO$	Checks filename for valid format.
UPDATE	Updates current UFD (PRIMOS II only).

DISPLY

Purpose

DISPLY updates the sense light settings according to argument A1. The bit values of A1 (1 = on, 0 = off) correspond to switch/light settings that are displayed on the computer control panel.

Usage

```
CALL DISPLY (A1)
```

Discussion

DISPLY is of use only on Prime computers that have lights on the control panel. Newer Prime computer models have no lights.

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.

ERRSET

Purpose

ERRSET sets ERRVEC, a system vector, then takes an alternate return or prints the message stored in ERRVEC and returns control to the system.

Usage

CALL ERRSET (altval, altrtn)

CALL ERRSET (altval, altrtn, messag, num)

CALL ERRSET (altval, altrtn, name, messag, num)

In Form 1, altval must have value 100000 octal and altrtn specifies where control is to pass. If altrtn is 0, the message stored in ERRVEC is printed and control returns to the system.

Forms 2 and 3 are similar; therefore, the arguments are described collectively as follows:

altval	A two-halfword array that contains an error code that replaces ERRVEC(1) and ERRVEC(2). altval(1) must not be equal to 100000 octal.
altrtn	A FORTRAN label preceded by a dollar sign. If altrtn is nonzero, control goes to altrtn. If altrtn is 0, the message stored in ERRVEC is printed and control returns to PRIMOS.
name	The name of a three-halfword array containing a six-letter word. This name replaces ERRVEC(3), ERRVEC(4), and ERRVEC(5). If name is not an argument in the call, ERRVEC(3) is set to 0.
messag	An array of characters stored two per halfword. A pointer to this messag is placed in ERRVEC(7).
num	The number of characters in messag. The value of num replaces ERRVEC(8).

Discussion

Refer to the description of PRERR, later in this chapter, for the contents of ERRVEC.

If a message is to be printed, first, six characters starting at ERRVEC(3) are printed at the terminal. Then the operating system checks to determine the number of characters to be printed. This information is contained in ERRVEC(8). The message to be printed is pointed to by ERRVEC(7). The operating system only prints the number of characters from the message (pointed to by ERRVEC(7)) that are indicated in ERRVEC(8). If ERRVEC(3) is 0, only the message pointed to by ERRVEC(7) is printed. The message stored in ERRVEC may also be printed by the PRERR command or the PRERR subroutine. The contents of ERRVEC may be obtained by calling subroutine GETERR.

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.

GETERR

Purpose

A user obtains ERRVEC contents through a call to GETERR.

Usage

```
CALL GETERR (xervec, n)
```

Discussion

GETERR moves n halfwords from ERRVEC into xervec.

On an Alternate Return

```
ERRVEC(1)  Error code.
ERRVEC(2)  Alternate value.
```

On a Normal Return

```
PRWFIL:
    ERRVEC(3) Record number.
    ERRVEC(4) Word number.

SEARCH:
    ERRVEC(2) File type.
```

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.

OVERFL

Purpose

Argument A1 in location AC5 is given a value of 1 if entry into F$ER was made; otherwise it is set to 2. F$ER is left in the no error condition. OVERFL is called to check if an overflow condition has occurred.

Usage

```
CALL OVERFL (A1)
```

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.

PHANT$

Purpose

PHANT$ starts a phantom user. This subroutine may be used only for non-CPL phantoms. It has been replaced with PHNTM$.

Usage

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

filnam	Name of command input file to be run by the phantom (integer array).
namlen	Length of characters of filnam (16-bit integer).
funit	File unit on which to open filnam. If funit is 0, unit 6 will be used (16-bit integer).
user	A variable returned as the user number of the phantom (16-bit integer).
code	The return code (16-bit integer). If it is 0, the phantom was initiated successfully. If code is E$NPHA, no phantoms were available. Other values of code are file system error indications.

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.

PRERR

Purpose

PRERR prints an error message on the user's terminal.

Usage

CALL PRERR

Example

A user wants to retain control on a request to open a unit for reading if the name was not found by SEARCH. To accomplish this, the program calls SEARCH and gets an alternate return. It then calls to GETERR and determines if an error occurred other than NAME NOT FOUND. To print the error message while maintaining program control, the user calls PRERR.

Discussion

ERRVEC consists of eight halfwords; their contents are as follows:

Word	Content	Remarks
ERRVEC(1)	Code	Indicates origin of error and nature of error.
ERRVEC(2)	Value	On alternate return, this is the value of the A-register. On normal return, this may have special meaning (refer to PRWFIL and EARCH error codes below).
ERRVEC(3) ERRVEC(4) ERRVEC(5) ERRVEC(6)	X X X X X X X X	ERRVEC(3), ERRVEC(4), and ERRVEC(5) contain a six-character filename of the routine that caused the error. (ERRVEC(6) is available for expansion of names.)
ERRVEC(7)	Pointer to message	For PRIMOS supervisor use.

Word	Content	Remarks
ERRVEC(8)	Message length	For PRIMOS supervisor use.

PRWFIL Error Codes

Code	Content	Remarks
PD	UNIT NOT OPEN	
PE	PRWFIL EOF (End of File)	Number of halfwords left (Information is in ERRVEC(2)).
PG	PRWFIL BOF (Beginning of File)	Number of halfwords left (Information is in ERRVEC(2)).

PRWFIL Normal Return

ERRVEC(3) Record number.

ERRVEC(4) Word number.

PRWFIL Read-Convenient

ERRVEC(2) Number of halfwords read.

SEARCH Error Codes

ERRVEC(1) Code, with one of the following values:

Code	Remarks
SA	SEARCH, BAD PARAMETER.
SD	UNIT NOT OPEN (truncate).
SD	UNIT OPEN ON DELETE.
SH	<Filename> NOT FOUND.

Code	Remarks
SI	UNIT IN USE.
SK	UFD FULL.
SL	NO UFD ATTACHED.
SQ	SEG-DIR-ER.
DJ	DISK FULL.

SEARCH Normal Return

ERRVEC (2) Type, with one of the following values:

Type	Remarks
0	File is SAM.
1	File is DAM.
2	Segment directory is SAM.
3	Segment directory is DAM.
4	UFD is SAM.

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.

SLITE

Purpose

Sets the sense light specified in argument A1 on or sets all sense lights off. If A1 = 0, all sense lights are reset off.

Usage

```
CALL SLITE (A1)
CALL SLITE (0)
```

Discussion

SLITE is of use only on Prime computers that have lights on the control panel. Newer Prime computer models have no lights.

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.

SLITET

Purpose

SLITET tests the setting of a sense light specified by the argument A1. The result of this test (1 = on, 2 = off) is in the location specified by the argument R.

Usage

```
CALL SLITET (A1,R)
```

Discussion

SLITET is of use only on Prime computers that have lights on the control panel. Newer Prime computer models have no lights.

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.

SSWTCH

Purpose

SSWTCH tests the setting of a sense switch specified by the argument A1. The result of this test (1 = set, 2 = reset) is stored in the location specified in argument R.

Usage

```
CALL SSWTCH (A1,R)
```

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.

TEXTO$

Purpose

TEXTO$ checks a filename for valid format. This subroutine has been replaced with FNCHK$.

Usage

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

filnam	An integer array containing the filename to be checked.
namlen	The length of filnam in characters (INTEGER*2).
trulen	An (INTEGER*2) set to the true number of characters in filnam. trulen is valid only if textok is .TRUE..
	trulen is the number of characters in filnam preceding the first blank. If there are no blanks, trulen is equal to namlen. See SRCH$$ for filename construction rules.
textok	A LOGICAL variable set to .TRUE. if filnam is a valid filename, otherwise set to .FALSE..

Caution

Names longer than 32 characters are truncated with no warning message.

Example

To read a name from the terminal, check for validity, and set trulen to the actual name length:

```
CALL I$AA12 (0, BUFFER, 80, $999)
CALL TEXTO$ (BUFFER, 32, TRULEN, OK) /* SET TRULEN
IF (.NOT. OK) GOTO <bad-name>
```

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.

UPDATE

Purpose

Under PRIMOS II, this subroutine updates the current UFD.

Usage

```
CALL UPDATE (key, 0)
```

key	Value must be 1 to update current UFD, send DSKRAT buffers to disk, if necessary, and undefine DSKRAT in memory (INTEGER*2).

Discussion

This call is effective only under PRIMOS II. Under PRIMOS it has no effect.

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.

APPENDIXES

A
Standard Conditions

The condition mechanism is described in Chapter 7. That description tells you how to signal conditions in general and how to handle them. It also defines the data structures associated with conditions.

This Appendix describes conditions raised by the operating system under various circumstances. These conditions are raised by PRIMOS or its associated utility software. Some other conditions not listed here are used by Prime software to communicate between different subsystems or different parts of a subsystem; normally the program is not affected by these conditions. If an ANY$ on-unit catches a condition not included in this Appendix, the condition should be ignored by returning from the on-unit.

In the list below, the meaning of each condition is given, followed by a description of the information available in the condition frame header structure produced by that condition.

The standard PL/I information structure is:

```
dcl 1 info based,
      2 file_ptr ptr options (short),  /* PL/I file control block */
      2 info_struct_len fixed bin,     /* Length in halfwords of */
                                       /* structure */
      2 oncode_value fixed bin,        /* Unique error code */
      2 ret_addr ptr options (short);  /* Points to statement */
                                       /* causing error. */
```

The data structures used by the condition mechanism are discussed in Chapter 7 under DATA STRUCTURE FORMATS.

In the descriptions below, software means that the machine state frame pointed to by cfh.ms_ptr is a condition frame header, and hardware means that this frame is a fault frame header. The notations cfh. and ffh. below refer to the condition frame header or fault frame header that is pointed to by cfh.ms_ptr or ffh.ms_ptr. The information structures referred to below are pointed to by cfh.info_ptr.

Unless otherwise noted below, the system default on-unit for each condition prints an appropriate diagnostic message on the user's terminal, terminates program execution, and returns to PRIMOS command level.

ACCESS_VIOLATION$

(hardware, returnable)

The process has attempted to perform a CPU instruction that has violated the access control rules of the processor. No information is readily available to differentiate between write violation, read violation, execute violation, and gate violation.

ffh.fault_type	Value '44'b3.
ffh.fault_addr	Contains the virtual address whose access is improper.
ffh.ret_pb	Points to the instruction causing the violation.

No information structure is available.

ALARM$

(software, returnable)

This condition is raised when the elapsed time watchdog timer expires. See the discussion of LIMIT$ in Chapter 8 for information on setting the elapsed time watchdog timer.

No information structure is available.

The default on-unit simply returns. This means that the expiration of the timer is ignored.

ANY$

(pseudo-condition)

An activation's on-unit for ANY$ is invoked if that activation does not have a specific on-unit for the condition that was raised. The condition frame header for the condition ANY$ describes the original condition directly; there is no separate condition frame header for the condition ANY$ unless ANY$ was explicitly raised by a call to SIGNL$ (not a recommended practice).

AREA

(software, not returnable)

This condition is raised when a storage area has been damaged, or when the target area for an attempted copy from one area to another was too small. Generally raised by PL1 only.

ARITH$

(hardware, returnable)

The process encountered an arithmetic exception fault.

ffh.fault_type	Value '50'b3.
ffh.fault_code	Hardware-defined exception code that partially identifies the cause of the fault.
ffh.ret_pb	Points to the next instruction to be executed upon return. There is no way in general to obtain a pointer to the faulting instruction.

No information structure is available.

The standard default handler for this condition resignals the appropriate arithmetic condition (SIZE, FIXEDOVERFLOW, etc.) with the appropriate information structure. This condition is raised by fixed overflow or underflow, or zero divide.

BAD_NONLOCAL_GOTO$

(software, not returnable)

The nonlocal GOTO processor was asked to transfer control to a label whose display (stack) pointer is invalid, or whose target activation has already been cleaned up. There is also a possibility that the user's stack may have been overwritten.

Information Structure:

```
DCL 1 info based,
      2 target_label label,
      2 ptr_to_nlg_call ptr,
      2 caller_sb ptr;
```

info.target_label	Label to which the nonlocal GOTO was attempted.
info.ptr_to_nlg_call	Pointer to the call to PL1$NL that requested this nonlocal GOTO.
info.caller_sb	Pointer to the activation (stack frame) requesting this nonlocal GOTO.

BAD_PASSWORD$

(software, not returnable)

This condition is raised by the procedures that change the user's attach point. It is caused by attempting to attach with an incorrect password to a directory requiring a password. This condition is signalled nonreturnable in order to increase the work function of machine-aided password penetration.

No information structure is available.

CLEANUP$

(software, returnable)

The nonlocal GOTO processor is in the process of invoking on-units for the condition CLEANUP$ in each activation on the stack, prior to actually unwinding the stack. The on-unit for this condition should return, unless it encounters a fatal error. Calls to CNSIG$ from a CLEANUP$ on-unit have no effect.

No information structure is available.

COMI_EOF$

(software, returnable)

End of file occurred on the command input file.

The default on-unit prints a diagnostic message and returns to the point of interrupt.

CONVERSION

(software, returnable)

This condition is raised when the source data for a data-type conversion contains one or more characters that are invalid for the target type. For example, nonnumeric characters appear in a character string that is to be converted to integer.

Information Structure: Standard PL/I information structure.

CPU_TIMER$

(software, returnable)

This condition is raised when the CPU watchdog timer expires. See the discussion of LIMIT$ in Chapter 8 for information on setting the CPU watchdog timer.

No information structure is available.

The default on-unit simply returns. This means that the expiration of the timer is ignored.

ENDFILE (file)

(software, returnable)

This condition is raised when an end of file is encountered while reading a PL/I file with PL/I I/O statements. The value of the ONFILE() built-in function identifies the file involved.

The standard PL/I condition information structure is provided. The value of info.oncode_value is undefined, and info.file_ptr identifies the file on which end of file occurred.

The default on-unit for this condition prints a diagnostic and then resignals the ERROR condition with an info.oncode_value of 1044.

ENDPAGE (file)

(software, returnable)

This condition is raised when end of page is encountered while writing a PL/I file using PL/I I/O statements. The value of the ONFILE() built-in function identifies the file on which the end of page was encountered.

The standard PL/I condition information structure is provided. The value of info.oncode_value is undefined; info.file_ptr identifies the file in question.

The default on-unit for this condition performs a PUT SKIP on the file, and then returns.

ERROR

(software, varies)

This condition is a catch-all error condition defined in PL/I. The default on-unit for most PL/I-defined conditions (such as KEY) results in the ERROR condition being resignalled. Hence, the programmer has the choice of handling a more- or less-specific case of the condition.

ERRRTN$

(software, not returnable)

A non-ring-0 call to the ring-0 entry ERRRTN was made, as the result of an ERRRTN SVC or a call to ERRPR$ with certain values of the key.

No information structure is available.

The default on-unit for this condition simulates a call to EXIT; hence, this condition should be signalled only while executing in a static-mode program.

EXIT$

(software, returnable)

The process has made a call to the EXIT primitive, via a direct call or an EXIT SVC. This condition should not be handled by user programs, since it is used by certain PRIMOS software to monitor the execution of static-mode programs.

No information structure is available.

The default on-unit for this condition simply returns.

FINISH

(software, returnable)

This condition is signalled before process termination, usually after files are closed. It closes any open files and returns to the point at which the condition was signalled. This condition is not signalled if the process is prematurely exhausted or destroyed. Available through PL/I. In PL/I, a STOP statement causes FINISH to be raised after files are closed. In this case, FINISH also raises the STOP$ condition.

The default on-unit simply returns.

FIXEDOVERFLOW

(hardware, not returnable)

This condition is detected by hardware and is raised when a fixed-point decimal or binary result is too large to fit into the hardware register or decimal field.

The standard PL/I condition information structure is provided.

HEAP_ERROR$

(software, non-returnable)

This condition is raised by user-class storage allocation and free routines to indicate that the memory structures defining the user's free memory area have become corrupted. See the discussion of STR$AU and STR$FU in Chapter 4.

No information structure is available.

The default on-unit prints a message informing the user about the corrupted storage area.

ILLEGAL_INST$

(hardware, returnable)

The process attempted to execute an illegal instruction.

ffh.fault_type	Value '40'b3.
ffh.ret_pb	Points at the faulting instruction.

No information structure is available.

ILLEGAL_ONUNIT_RETURN$

(software, not returnable)

An on-unit for a condition attempted to return, but returning was disallowed by the procedure that raised the condition.

Information Structure: The standard-format condition frame header that describes the condition whose on-unit illegally attempted to return.

ILLEGAL_SEGNO$

(hardware, returnable)

The process referenced a virtual address whose segment number is out of bounds.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	The virtual address that is in error.

No information structure is available.

KEY (file)

(software, returnable)

The KEY condition is raised when reading or writing a keyed PL/I file with PL/I I/O statements, and the supplied key does not exist (READ) or already exists (WRITE). The value of the ONFILE() built-in function identifies the file in question; the value of the ONKEY() built-in function contains the key in error.

Information Structure: The standard PL/I condition information structure. The value of info.oncode_value is undefined; the value of info.file_ptr identifies the file in question.

The default on-unit prints a diagnostic and resignals the ERROR condition, with an info.oncode_value of 1045.

LINKAGE_FAULT$

(hardware, returnable)

The process referenced through an indirect pointer (IP) that is a valid unsnapped dynamic link, but the desired entrypoint could not be found in any of the dynamic link tables.

ffh.fault_type	Value '64'b3.
ffh.fault_addr	Points to the faulting indirect pointer.
ffh.ret_pb	Points to the faulting instruction.

Information Structure:

```
DCL 1 info based,
      2 entry_name char(32) var;
```

info.entry_name	Name of the entry point that could not be found.

LISTENER_ORDER$

(software, varies)

This condition is used internally by the command loop to manage its recursion. Users should never make on-units for this condition, and user default on-units (ANY$) should always pass this condition on by returning.

LOGOUT$

(software, returnable)

This condition is raised when a user or the operator is trying to force log out a process.

Information Structure:

```
DCL 1   logout_info,
        2  reason fixed;     /* reason for logout;
                               codes available in PRIMOS source */
```

The default on-unit logs out the process. When LOGOUT$ is signalled, the intercepting process has between one and two minutes to do its cleanup before being force-logged out.

NAME

(software, returnable)

This condition occurs only during data-directed input. It occurs when stream assignment in a GET statement is read whose variable does not match the variable name in the data list. After execution of the on-unit, the process returns to the data-directed input as if the invalid input were processed. Generally raised by PL/I only.

NO_AVAIL_SEGS$

(hardware, returnable)

The process referenced a virtual address that refers to a segment that has not yet been created. At the moment, the system has no free page tables to assign to the segment. If the on-unit for this condition returns, the reference is retried. If, in the meantime, this or some other process deleted a segment, the reference now has the possibility for successful completion.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	Virtual address that is causing the attempted segment creation.

No information structure is available.

NONLOCAL_GOTO$

(software, returnable)

This condition is signalled by the PL/I nonlocal GOTO processor PL1$NL just prior to setting up the stack unwind (and hence prior to the invocation of any CLEANUP$ on-units). This condition exists to enable certain overseer software (such as the debugger) to be informed that the nonlocal GOTO is occurring. The default handler for this condition simply returns. When a procedure handling this condition wishes to let the nonlocal GOTO occur, it should simply return (without continue-to-signal set).

Information Structure: Same as for the BAD_NONLOCAL_GOTO$ condition.

NPX_SLAVE_SIGNALED$

(software, not returnable)

A condition was raised in your slave process running on some remote system. The following message is printed:

```
Condition signalled in NPX slave on nodename
ERROR: Condition "condition name" raised at segment no./
       halfword no.
```

Information Structure:

```
DCL  1  npx_slave_info,
        2  node fixed,    /* npx node number on which
                            slave is running */
        2  orig_condition char (32) var,  /* condition
                                            raised in slave */
        2  orig_info_data (129) fixed;  /* info
                                          structure from slave */
```

When the slave detects a signalled condition, it transmits to the master, which signals the condition NPX_SLAVE_SIGNALED$. Its result is the printout of the message shown above. The slave transmits to the master all types of conditions signalled except the following:

EXIT$

FINISH

LINKAGE_FAULT$

NONLOCAL_GOTO$

REENTER$

STRINGSIZE

These conditions are handled differently by the slave's on-unit. They are returned without transmitting to the master; that is, the master side will not get the condition NPX_SLAVE_SIGNALED$.

NULL_POINTER$

(hardware, returnable)

The process referenced through an indirect pointer or base register whose segment number is '7777'b3. This is considered to be a reference through a null pointer, although user software should always employ the single value '7777/0 for the null pointer.

ffh.fault_type	Value '60'b3.
ffh.ret_pb	Points to the faulting instruction.
ffh.fault_addr	Null pointer through which a reference was made.

No information structure is available.

The default on-unit for this condition resignals the ERROR condition with the appropriate information structure.

OUT_OF_BOUNDS$

(hardware, returnable)

The process referenced a page of some segment that cannot be referenced in any ring (that is, no main memory or backing storage is allocated for that page, and allocation is not permitted).

ffh.fault_type	Value '10'b3.
ffh.ret_pb	Points at the faulting instruction.
ffh.fault_addr	The offending virtual address.

No information structure is available.

OVERFLOW

(hardware, not returnable)

This condition is raised when the result of a floating-point binary calculation is too large for representation. It may occur within a register or as a store exception. The default on-unit prints a message and signals the ERROR condition. User on-units may not return to the point of interrupt. However, if the default on-unit is invoked, and if the user types START, the register or memory location affected is set to the largest possible single-precision floating-point number, and calculation continues.

PAGE_FAULT_ERR$

(hardware, returnable)

The process encountered a page fault referencing a valid virtual address, but, due to a disk error, the page control mechanism was not able to load the page into main memory. If the on-unit for this condition returns, the reference is retried, with some likelihood that the disk read will succeed.

ffh.fault_type	Value '10'b3.
ffh.ret_pb	Points at the faulting instruction.
ffh.fault_addr	Virtual address, the page for which cannot be retrieved.

No information structure is available.

PAUSE$

(software, returnable)

The process executed a PAUSE statement in a FORTRAN program. This condition should not be handled by user programs since it is used by Prime software to ensure the proper operation of the FORTRAN PAUSE statement.

No information structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

PH_LOGO$

(software, returnable)

This condition is raised when a phantom that you spawned is logging out.

No information structure is directly available. Use the subroutine LON$R, described in Chapter 5, to obtain information about the phantom.

POINTER_FAULT$

(hardware, returnable)

This is the process referenced through an indirect pointer (IP) whose fault bit is on, but that pointer is not a valid unsnapped dynamic link. This error condition is frequently caused by making a subroutine call with too few arguments. The condition is raised when the called subroutine attempts to access one of its arguments through a faulted pointer.

ffh.fault_type	Value '64'b3.
ffh.fault_addr	Points to the faulting indirect pointer.
ffh.ret_pb	Points to the faulting instruction.

No information structure is available.

QUIT$

(hardware, software, returnable)

The user actuated QUIT (BREAK key or CONTROL-P) on the terminal.

If this is a hardware signal, then ffh.fault_type has the value '04'b3.

cfh.ret_pb or ffh.ret_pb points to the next instruction to be executed in the faulting procedure.

No information structure is available.

The default on-unit flushes the input and output buffers of the user's terminal, prints the message "QUIT." on the terminal, and calls a new command level.

RECORD

(software, returnable)

This condition is raised when record size is different from the variable defined in the PL/I source. Generally raised by PL/I only.

REENTER$

This condition is raised by the PRIMOS REENTER (REN) command and reenters a subsystem that has been temporarily suspended due to another condition (such as a QUIT$ signal).

If the interrupted operation can be aborted, the subsystem's on-unit can accomplish this by performing a nonlocal GOTO back into the subsystem at the appropriate point.

If the QUIT$ occurred during an operation that must be completed, the on-unit should set the info.start_sw to '1'b, record the QUIT$ request within the subsystem, and return. The REN command then executes a START command which restarts the subsystem at the point of interrupt. When the operation is complete, the subsystem should then honor the recorded QUIT$ request.

The default on-unit returns without setting the info.start_sw. The REN command then prints a diagnostic and returns since it assumes the stack held no subsystem able to accept reentry.

Information Structure:

```
DCL 1 info based,
      2 start_sw bit(1) aligned;
```

RESTRICTED_INST$

(hardware, returnable)

The process attempted to execute an instruction whose use is restricted to ring-0 procedures. Certain of these instructions (in the I/O class) can be simulated by ring 0. An instruction that causes this condition to be raised could not be simulated by this mechanism.

ffh.fault_type	Value '00'b3.
ffh.ret-pb	Points to the faulting instruction.

R0_ERR$

(software, returnable)

A ring-0 call to ERRPR$ or ERRRTN was made, as the result of a detected fatal error condition.

No information structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

SIZE

(software, not returnable)

This condition is raised when a program tries to do an arithmetic conversion and the value is too large to fit into the target data type. It can occur when converting a floating-point number, a decimal integer, or a character string.

The standard PL/I condition information structure is provided.

STACK_OVF$

(hardware, returnable)

The process overflowed one of its stack segments, but the condition mechanism was able to locate a stack on which to raise this condition.

ffh.fault_type	Value '54'b3.
ffh.fault_addr	The last stack segment in the chain of stack segments of the stack that overflowed. It is this segment that contains the zero extension pointer that caused the stack overflow fault.
ffh.ret_pb	Points to the faulting instruction.

No information structure is available.

STOP$

(software, not returnable)

The process executed a STOP statement in a higher-level-language program. This condition should not be handled by user programs, as it is used by Prime software to ensure the proper operation of the STOP statement in the various languages.

No information structure is available.

The default on-unit for this condition performs a nonlocal GOTO back to the command processor that invoked the procedure (or one of the dynamic descendants) that executed the STOP statement.

STORAGE

(software, returnable)

The STORAGE condition indicates there is insufficient memory to satisfy a request to allocate dynamic memory. In PL1G, the condition can be raised either through the ALLOCATE statement or by the compiler making its own call.

The standard PL/I condition information structure is provided.

STRINGRANGE

(software, returnable)

One argument of the PL/I SUBSTR function is out of range of the string.

STRINGSIZE

(software, returnable)

The target of a string assignment is too small to contain the value. The default on-unit simply returns.

Information Structure: The standard PL/I condition information structure is provided.

SUBSCRIPTRANGE

(software,returnable)

A subscript is out of range.

Information Structure: Standard PL/I information structure.

SUBSYS_ERR$

The subroutine SS$ERR raises this condition when it is called by a subsystem that is not interactive (that is, one run by a CPL or command file). The default on-unit for SUBSYS_ERR$ aborts execution of the subsystem and forces the severity code to have a positive sign. Any command input file is aborted.

SVC_INST$

(hardware, returnable)

The process executed an SVC instruction, but the system was not able to perform the operation. If the user is in "SVC virtual" mode, all SVC instructions result in this condition being raised.

ffh.fault_type	Value '14'b3.
ffh.ret_pb	Points to the location following the SVC instruction.

Information Structure:

```
DCL 1 info based,
      2 reason fixed bin;
```

info.reason values are:

1 Bad SVC operation code or bad argument(s).

2 Alternate return needed but was 0.

3 Virtual SVC handling is in effect in this process.

For the case of virtual SVCs only (info.reason code of 3), the static-mode default on-unit simulates the Prime 300 fault handling for the SVC fault, if the appropriate halfword of segment '4000 is nonzero. If this halfword is 0 or if there is no static-mode program in execution, the standard default handler prints a diagnostic and calls a new command level. (See the System Architecture Reference Guide for the exact location.)

SYSTEM_STORAGE$

This condition is raised when one of the routines managing process-class dynamic memory detects an error. These routines are described in Chapter 4.

The default on-unit initializes the command environment.

TRANSMIT

(software, returnable)

This condition occurs when data cannot be transmitted reliably between a data set and PL/I storage.

UII$

(hardware, returnable)

The process executed an unrecognized instruction that nevertheless caused an unimplemented instruction fault, or else the system UII handler detected an error in processing the valid UII.

The fault frame header that accompanies this condition is nonstandard in that ffh.regs is not valid. The registers at time of fault are unavailable.

ffh.ret_pb	Points to the next instruction to be executed in the faulting procedure.

UNDEFINEDFILE (file)

(software, not returnable)

This condition is raised when an OPEN statement cannot associate an input file with an existing PRIMOS file or device. The default on-unit prints a message and signals the ERROR condition.

UNDEFINED_GATE$

(software, not returnable)

This condition is signalled when the process called an inner-ring gate segment at an address within the initialized portion of the gate segment, but there was no legal gate at that address. This error can arise because gate segments are padded with illegal gate entries, from the last valid gate entry to the next page boundary, and the program has attempted to construct and use a pointer into the gate segment, instead of using the dynamic linking mechanism.

No information structure is available.

UNDERFLOW

(hardware, returnable)

This condition is signalled when the result of the floating-point binary or decimal calculation is too small for representation. The default on-unit sets the floating-point accumulator to 0.0e0. If the underflow occurred as a store exception, the affected portion of memory is also set to 0.0e0. The default on-unit returns and the calculation proceeds, using the 0.0e0 value.

The standard PL/I condition information structure is provided.

WARMSTART$

(software, returnable)

This condition is raised for every process when the operator successfully performs a warm start. The default on-unit prints the following message and returns:

```
***** WARM START *****
```

No information structure is available.

ZERODIVIDE

(hardware, not returnable)

This condition is signalled when a division by 0 (floating-point or fixed-point) occurs. The default on-unit prints a message and signals the ERROR condition. For compatibility with earlier versions of PRIMOS, if the condition is the result of a floating-point operation, the user may type START following the printing of the message. The default on-unit then sets the register involved to the largest possible single-precision floating-point value and proceeds with the calculation.

The standard PL/I condition information structure is provided.

B
Data Type Equivalents

In order 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, CBL, COBOL, 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.

Note

The term PL/I refers both to full PL/I and to PL/I Subset G (PL/I-G).

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. See the section How to Set Bits in Arguments in Chapter 1.

Table B-1
Data Type Equivalents

Generic Unit	BASIC/ VM	C	CBL	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL/I
Bit string		unsigned int					SET	BIT(n)
Byte string (Max. 32767)	INT	char	DISPLAY PIC A(n) PIC X(n) FILLER	DISPLAY(5) PIC A(n) PIC X(n) PIC 9(n)	INTEGER	CHARACTER *n	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)
Byte string (1 digit per byte)			DISPLAY PIC 9(n)					PICTURE
Byte string (2 digits per byte)			COMP-3					FIXED DECIMAL
Varying character string							STRING[n]	CHAR(n) VARYING
32 bits (Two halfwords)		Pointer (32IX-mode)						POINTER OPTIONS (SHORT)
48 bits (Three halfwords)		Pointer (64V-mode)					Pointer	POINTER
Variable-length array		Array					Conformant array	Star-extent array

Table B-1 (continued)
Data Type Equivalents

Generic Unit	BASIC/ VM	C	CBL	COBOL	FORTRAN IV	FORTRAN 77	PASCAL	PL/I
16 bits (Halfword)	INT	short enum	COMP PIC S9(1)- PIC S9(4)	COMP	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER Enumerated	FIXED BIN FIXED BIN(15)
32 bits (Word)	INT*4	int long	COMP PIC S9(5)- PIC S9(9)		INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	FIXED BIN(31)
64 bits (Double word)			COMP PIC S9(10)- PIC S9(18)					
32 bits (Float single precision)	REAL	float	COMP-1		REAL REAL*4	REAL REAL*4	REAL	FLOAT BIN FLOAT BIN(23)
64 bits (Float double precision)	REAL*8	double	COMP-2		REAL*8	REAL*8	LONGREAL	FLOAT BIN(47)
128 bits (Float quad precision)						REAL*16		
1 bit		struct{ unsigned; }name;						BIT BIT(1)
Bit aligned (Halfword)							BOOLEAN	BIT(1) ALIGNED

Note: For a discussion of how to implement the Generic Units in PMA, see the PMA INTERFACE chapter in Volume I of the Subroutines Reference Guide.

C
File-system Date Format

Some of the routines in this volume refer to "File-system Date Format" (or FS-date). This is a 32-bit value that is used by the PRIMOS filing system to record date and time information.

A date and time in File-system Date Format occupies 32 bits, so it may be held in a fullword integer (FORTRAN INTEGER*4). The format is designed so that times can be compared arithmetically with correct results. For example, if date1 and date2 are two 32-bit integers, and date1 is less than date2, then the time represented by date1 is earlier than the time represented by date2. (Integer comparison of two dates does not work if they fall on opposite sides of 1 Jan 1964, because the high order bit of year is the arithmetic sign of the integer. It becomes a 1 on that date, changing the sign of the integer.)

The time is accurate to the nearest four seconds. The word quadsecond has been invented to stand for a unit of time of four seconds. This unit was chosen so that the time field will is positive. The routines CV$DQS, CV$DTB, CV$FDA, CV$FDV, and CV$QSD, described in Chapter 6, are provided to convert between File-system Date Format and other, more convenient formats.

The date is encoded as three integers packed into the first 16 bits, as described in the following structure:

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

year	Year number, minus 1900. For example, 86 represents the year 1986, and 117 represents the year 2017.
month	Month, from 1 for January to 12 for December.
day	Day of the month, from 1 to 31.
quadseconds	Number of quadseconds (groups of four seconds) elapsed since midnight of the date described by the above three fields.

INDEXES

Index of Subroutines by Name

Index

S

SURVEY

READER RESPONSE FORM

DOC10082-1LA Subroutines Reference Guide Volume III

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™

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

READER RESPONSE FORM

DOC10082-1LA Subroutines Reference Guide Volume III

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™

Attention: Technical Publications
Bldg 10

READER RESPONSE FORM

DOC10082-1LA Subroutines Reference Guide Volume III

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™

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