Prime®

Subroutines Reference III: Operating System

Revision 23.0

DOC10082–2LA

Subroutines Reference III: Operating System

Second Edition

Glenn Morrow

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

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

Printing History

First Edition (DOC10082–1LA) August 1986 for Revision 20.2
Update 1 (UPD10082–11A) July 1987 for Revision 21.0
Update 2 (UPD10082–12A) August 1988 for Revision 22.0
Update 3 (in RLN10247–1LA) July 1989 for Revision 22.1
Second Edition (DOC10082–2LA) June 1990 for Revision 23.0

Credits

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

How to Order Technical Documents

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

- United States customers call Prime Telemarketing, toll free, at

  1-800-343-2533

  Monday through Thursday, 8:30 a.m. to 8:00 p.m., and
  Friday, 8:30 a.m. to 6:00 p.m. (EST).

- International customers contact your local Prime subsidiary or distributor

PRIME SERVICE℠

To obtain service for Prime systems

- United States customers call Prime Customer Support Center, toll free, at

  1-800-800-PRIME

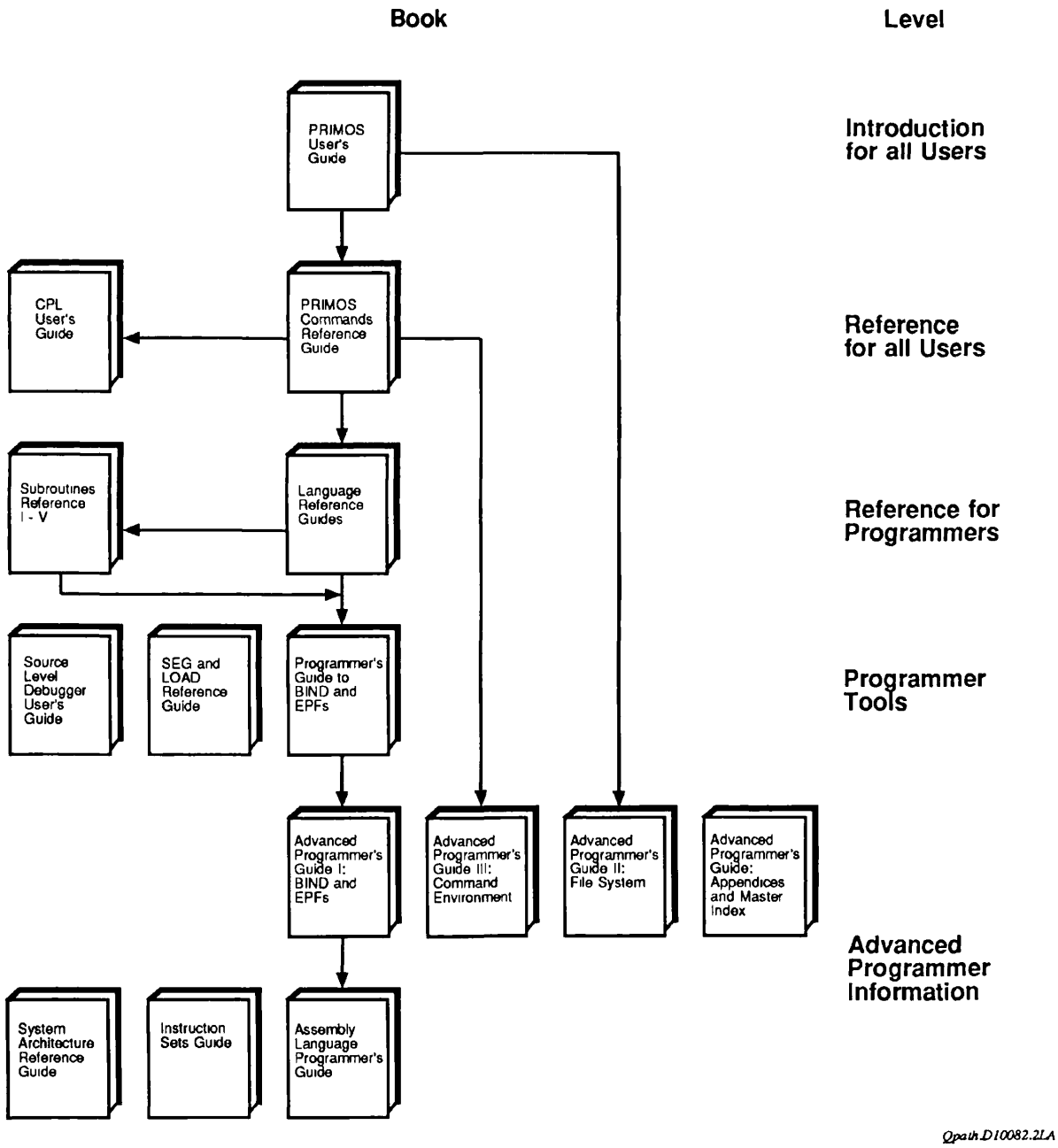
- International customers contact your Prime representative.

Surveys and Correspondence

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

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

Reading Path for PRIMOS Documentation



Contents

5 Program Control

6 Conversion Routines and Other Utilities

7 Condition Mechanism

8 Semaphores and Timers

9 Message Facility

Appendices

Indexes

Index of Subroutines by Function

About This Book

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

Overview of This Series

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

Volume I: Using Subroutines

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

Volume II: File System

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

Volume III: Operating System

Volume III describes system subroutines. The subroutines covered in this volume are general system calls to the operating system and the standard system

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

Volume IV: Libraries and I/O

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

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

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

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

Volume V: Event Synchronization

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

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

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

Specifics of This Volume

This volume of the *Subroutines Reference* series presents detailed descriptions of system subroutines.

Chapter 2 describes subroutines for general operating system information, metering information, and information specific to the current user. This chapter includes several additions for Rev. 23.0, including a new subroutine, DS$SEND_CUSTOMER_UM, for sending messages to the DSM server, and extensive additions to the G$METR subroutine.

Chapter 3 describes subroutines that perform input and output on the user's main terminal, as well as procedures for controlling terminal interaction. *Subroutines Reference IV: Libraries and I/O* describes routines for other types of input and output.

Chapter 4 describes subroutines that enable allocation and freeing of blocks of contiguous memory. It also contains specific routines related to the use of EPF functions, and procedures that tell how much memory is available. *Subroutines Reference II: File System* describes the other routines used for manipulating EPFs.

Chapter 5 contains subroutines that control the user's command environment and terminate programs. Routines for serialization and for controlling static-mode programs and phantom processes also are described in this chapter.

Chapter 6 contains subroutines that convert data from one form to another and that manipulate data. It includes descriptions of subroutines that perform binary searches, encrypt passwords, and store and retrieve characters in arrays.

Chapter 7 describes subroutines that implement the condition mechanism.

Chapter 8 describes subroutines that manipulate semaphores, signal the completion of a timed interval, and cause a specific delay before processing is resumed. *Subroutines Reference V: Event Synchronization* describes another, independent facility for posting and receiving notices and signalling timed intervals.

Chapter 9 describes subroutines used to implement the PRIMOS® message facility for exchanging messages between users. *Subroutines Reference V: Event Synchronization* describes another, independent facility for message exchange.

The appendices provide a complete listing of PRIMOS standard conditions, a chart of data type equivalents for different languages, an explanation of the Prime file-system date format, and reference material for subroutines that are obsolete or superseded, but still supported.

Three indexes enable the reader to find information quickly. These are

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

Suggested References

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

- *Subroutines Reference I: Using Subroutines* (DOC10080-2LA) and its update for Rev. 23.0 (UPD10080-21A)
- *Subroutines Reference II: File System* (DOC10081-2LA)
- *Subroutines Reference IV: Libraries and I/O* (DOC10083-2LA)
- *Subroutines Reference V: Event Synchronization* (DOC10213-1LA) and its update for Rev. 23.0 (UPD10213-11A)

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

The *PRIMOS User's Guide* (DOC4130-5LA) contains information on system use, directory structure, the condition mechanism, CPL files, ACLs, global variables, and how to load and execute files with external subroutines. New information for Rev. 23.0 can be found in the *PRIMOS User's Release Document* (DOC10316-1PA).

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

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

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

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

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

The following related Prime publications are also available.

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

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

Prime Documentation Conventions

The following conventions are used throughout this document. The examples in the table illustrate the uses of these conventions.

Convention	*Explanation*	*Example*
Uppercase	In subroutine syntax or text, words in uppercase indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	**DATE$**
Italic	Variables in subroutine syntax, text, or messages are indicated by lowercase italic.	**ASSUR$ (*desired_time*);**
Abbreviations	If a subroutine has an abbreviation, the abbreviation is placed immediately below the full form.	**TMR$GTIM** **TMR$TM**
Brackets	Brackets enclose a list of one or more optional items. Choose none, one, or several of these items.	**LD** [**-BRIEF** / **-SIZE**]
Braces	Braces enclose a list of items. Choose one and only one of these items.	**CLOSE** { *filename* / **-ALL** }
Braces within brackets	Braces within brackets enclose a list of items. Choose either none or only one of these items; do not choose more than one.	**BIND** [{ *pathname* / *options* }]
Monospace	Identifies screen output, user input, prompts, and messages.	`address Connected`
User input in examples	In examples, user input is underscored but system prompts and output are not.	`OK, RESUME MY_PROG`
Hyphen	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	**SPOOL -LIST**
Subscript	A subscript after a number indicates that the number is not in base 10. For example, the subscript 8 is used for octal numbers.	200_8
Parentheses	In command or statement formats, you must enter parentheses exactly as shown.	**DIM** *array (row, col)*

Overview of Subroutines

1

A **subroutine** is a module of code that can be called from another module. It is useful for performing operations that cannot be performed by the calling language, or for performing standard operations faster. Users can write their own subroutines to supply customized or repetitive operations. However, this manual 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 *Subroutines Reference I: Using Subroutines*.

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 modules.

Subroutine Descriptions

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

- The name of the subroutine and, optionally, a six-character alias for use in FTN programs. This is followed by a brief description of what the subroutine does.
- **Usage.** The format of the 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.

READY$

Subroutines Reference III Operating System

READY$

READY$ displays the PRIMOS command-level prompts

Usage

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

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

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

typecode
INPUT Prompt type 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 *PRIMOS 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

2-40 Second Edition

101.01.D10082.2LA

Figure 1–1. A Subroutine Description

Subroutine Usage

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

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

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

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, rather than 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 *buffer* must be a character string, while *char_count* and *actual_count* 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 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. *Subroutines Reference I: Using Subroutines* 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 two bytes (one halfword) of storage for a CHAR(*n*) VAR string contain an integer that specifies the string length; these are followed by the string, one character per byte.
CHAR(*) VAR	Also CHARACTER(*) VARYING. Specifies a character string or array whose length is unknown at the time of declaration. The first two bytes (one halfword) of storage for a CHAR(*) VAR string contain an integer that specifies the string length; these are followed by the string, one character per byte.
FIXED BIN	Also FIXED BINARY, BIN, FIXED BIN(15). Specifies a 16-bit (halfword) signed integer.
FIXED BIN(31)	Specifies a 32-bit signed integer.
(*n*) FIXED BIN	An integer array of *n* elements. See below for more information about arrays.
FLOAT BIN	Also FLOAT BIN(23), FLOAT. Specifies a 32-bit (one-word) floating-point number.
FLOAT BIN(47)	Specifies a 64-bit (double-word) floating-point number.
BIT(1)	Specifies a 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 10-element array of integers. The keywords FIXED BIN, however, can be replaced by any data type. In PL/I, by default, arrays are indexed starting with the subscript 1; the first integer in this array is ITEMS(1).

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

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

WORD is an array indexed from 0 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 the 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* 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.

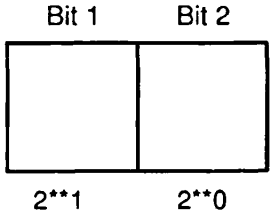
The Usage section of the subroutine descriptions gives both the function invocation and the subroutine invocation for subroutines that are likely to be called in both ways.

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.

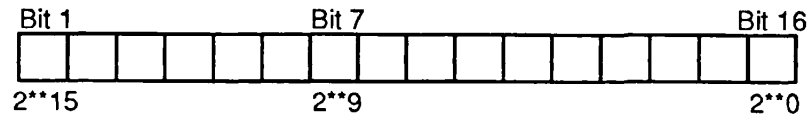


101.02.D10082.2LA

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

The **low-order** bit is in the position of 2 to the 0 power, and its value, if 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.



101.03.D10082.2LA

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

Key Names as Arguments

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

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 *Subroutines Reference I: Using Subroutines* 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 *Subroutines Reference I: Using Subroutines.*

Standard Error Codes

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

Standard error codes are of the form E$*xxxx*, where *xxxx* is any combination of letters. For example, the error code

E$DVIU

corresponds to the error message `The device is in use.`

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

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

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

Libraries and Addressing Modes

The *Subroutines Reference* document set 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 1 of *Subroutines Reference I: Using Subroutines* 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 with BIND, SEG, or LOAD (the R-mode loader).

This is accomplished by loading a binary library supplied by Prime using the LI (for Library) command. The Loading and Linking Information section under each 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 EPF libraries supplied by Prime, SYSTEM_LIBRARY.RUN or PRIMOS_LIBRARY.RUN. The pathnames to these libraries must be in the system search rules.

Because many of the subroutines 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 *Subroutines Reference I: Using Subroutines.*

Core Operating System Services

2

This chapter describes subroutines that provide core operating system services to the programmer.

The first part of this chapter presents subroutines involving general operating system information. This part includes the subroutine DS$SEND_CUSTOMER_UM, new at Rev. 23.0, that allows you to send messages to the DSM server.

The second part of this chapter describes subroutines involving system information specific to the current user.

The third part of this chapter describes system status and metering subroutines. The subroutines whose names begin with DS$ are intended to support System Information and Metering (SIM) commands. They are being made available to programmers because they provide potentially useful information. Similarly, the routine G$METR is provided to support the USAGE command, but may be of more general use.

System Information Routines

This section describes the following subroutines:

Routine	*Function*
ABSW	Return cold-start setting of ABBREV switch.
CKDYN$	Determine if routine is dynamically accessible.
CL$MSG	Return text of a specified system prompt.
CPUID$	Return model number of Prime computer.
DATE$	Return current date and time.
DS$SEND_CUSTOMER_UM	Send a message to DSM.
ER$TEXT	Return text representation of error code for specified PRIMOS subsystem.
GINFO	Return PRIMOS II information.
GSNAM$	Return current PRIMOS system name.
LOV$SW	Indicate if login-over-login permitted.
PRI$RV	Return operating system revision number.
RSEGAC$	Determine access to a segment.
SNCHK$	Check validity of system name passed to it.
USER$	Return user number and count of users.

ABSW

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$

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_name
INPUT. The name of the dynamic entrypoint. Specify this name in uppercase letters. If you specify it in lowercase letters, or in a mixture of uppercase and lowercase, CKDYN$ may fail to find the entrypoint.

code
OUTPUT. Standard error code. Possible values are

E$OK	Dynamic entrypoint *routine_name* was found.
E$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.

CL$MSG

This routine returns the text of a specified command line system prompt.

Usage

```
DCL CL$MSG ENTRY (FIXED BIN, FIXED BIN, FIXED BIN,
                  CHAR(*)VAR, FIXED BIN);

CALL CL$MSG (key, in_code, msg_size, msg, code);
```

Parameters

key

INPUT. Specify one of the two following keys, to select either the long or the brief form of the system prompt.

K$LONG	Long form
K$BRIEF	Brief form

in_code

INPUT. Specify a numeric value as indicated below to select one of the three possible types of system prompt (error, warning, or ready).

> 0	Error prompt
< 0	Warning prompt
0	Ready prompt

msg_size

INPUT. Specify the maximum size in characters of the buffer specified in the *msg* parameter (see below).

msg

OUTPUT. A buffer that receives the text of the system prompt.

code

OUTPUT. The error code. Among the possible values are

E$OK	No error.
E$BKEY	An invalid key is specified in *key*.

E$BFTS	The maximum size of the prompt, as specified in *msg_size*, is too small. The text returned to *msg* is truncated to the number of characters specified in *msg_size*.
E$BLEN	A negative value is specified for *msg_size*. *msg_size* must be positive.

Discussion

As specified by the caller, CL$MSG returns the text, in long or brief form, of the system's command line ready prompt, warning prompt, or error prompt. The system issues a ready prompt after the successful execution of a PRIMOS command. It issues a warning prompt after an error occurs during execution of a command that does not prevent execution from completing. It issues an error prompt after an error occurs that prevents execution from completing.

By default, the ready, warning, and error prompts are `'OK, '`, `'OK, '`, and `'ER! '`, respectively. You can specify alternatives to these defaults using the RDY command. For information about the RDY command, see the *PRIMOS Commands Reference Guide*.

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

CPUID$

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

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, /* Must be 1. */
   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 FIXED BIN(31),
   2  res4 FIXED 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

E$OK	No error
E$BPAR	*version* is not 1

Discussion

At Rev. 20.2 and later revisions, the following values can be returned in *cpu_model*:

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

Value	*Processor Model*
10	P550-II
11	P2250
15	P9950
16	P9650
17	P2550
18	P9955
19	P9750
21	P2350
22	P2655
23	P9655
25	P2450
26	P4050
27	P4150
28	P6350
29	P6550
30	P9955-II
31	P2755
32	P2455
33	P5310
34	P9755
35	P2850
36	P2950
37	P5330
38	P4450
39	P5370
40	P6650
41	P6450

Value	*Processor Model*
43	P5320
44	P5340

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$

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. The current date in file–system date format (FS-date).

Discussion

DATE$ returns the current date and time in the standard bit-encoded FS-date format. The FS-date format 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.

DS$SEND_CUSTOMER_UM

This routine enables a customer product to send an unsolicited message to the Distributed System Management (DSM) server.

Usage

```
DCL DS$SEND_CUSTOMER_UM ENTRY (CHAR(*)VAR,
                               FIXED BIN,
                               CHAR(*)VAR,
                               FIXED BIN,
                               FIXED BIN);

CALL DS$SEND_CUSTOMER_UM (text, severity, product_name,
                          reserved, code);
```

Parameters

text

INPUT. The text of the unsolicited message. Limited to 1024 characters.

severity

INPUT. The severity of the message. The following are valid severity codes

DS$SECURITY_VIOLATION	0
DS$ALARM	1
DS$WARNING	2
DS$INFORMATION	3
DS$FAILURE	4
DS$DIAGNOSTIC	5
DS$ACCOUNTING	6
DS$STATISTIC	7
DS$APPLICATION_DATA	8

These severity codes are defined in the insert file SYSCOM>DS$SEVERITY_KEYS.INS.PL1.

product_name

INPUT. The customer product's name as registered with DSM. You can specify this name in either uppercase or lowercase letters. If the product has not been registered, this subroutine returns an error code.

reserved

INPUT. Reserved for future use. Must be zero.

code

OUTPUT. The return status codes, as defined in the nonstandard error file SYSCOM>DS$ERROR_KEYS.INS.PL1. Possible values are

DS$OK	Message sent successfully.
DS$ER_UNKNOWN_CUST_PROD	The product specified is not registered with DSM as a customer product.
DS$ER_TEXT_TOO_LONG	Text length > 1024 characters.
DS$ER_BAD_TEXT_LENGTH	Text length < 0 characters.
DS$ER_BAD_SEVERITY	Severity is not a valid severity.
DS$ER_DSM_UNAVAILABLE	DSM is not running on the local node.
DS$ER_INSUFFICIENT_RESOURCES	There are insufficient system resources available to process this unsolicited message. There could be no virtual circuits available or no memory available. The queue to the DSM server could be full or there could be no ISC buffers available. You may need to restart DSM.
DS$ER_INTERNAL_ERROR	DSM has been unable to send the product's unsolicited message for an unspecified reason. Report any occurrence of this error to the System Administrator.

Discussion

The DS$SEND_CUSTOMER_UM subroutine permits a customer product to send an unsolicited message to DSM. This unsolicited message is handled by the DSM Unsolicited Message Handler (UMH), as described in the *DSM User's Guide*.

Before calling this subroutine, the name of the customer product must be registered with DSM. Your System Administrator should use the CONFIG_DSM command to register the product name, then restart DSM with

the new configuration. The System Administrator should also use the CONFIG_UM command to set up routing for the product's unsolicited messages. If no routing has been specified for a customer product, all of its unsolicited messages are sent to the DSM default log. These commands are further described in the *DSM User's Guide.*

Loading and Linking Information

V-mode and I-mode: Use the library DSMLIB.BIN from the LIB directory.

R-mode: Not available.

Effective for PRIMOS Rev. 23.0 and subsequent revisions.

ER$TEXT
ER$TXT

This routine returns error message text for PRIMOS and specified PRIMOS subsystems.

Usage

```
DCL ER$TEXT ENTRY (CHAR(*)VAR, FIXED BIN(15),
                   CHAR(*)VAR);

CALL ER$TEXT (sscode, ercode, ertext);
```

Parameters

sscode

INPUT. The subsystem for which ER$TEXT is to return an error message. Specify a code that designates a PRIMOS subsystem. Subsystem codes are defined in the file ERRORMSGHDLR.INS. *language*, where the suffix *language* denotes the programming language. If the subsystem code is invalid, ER$TEXT returns a null string. Some possible subsystem codes are

SSC$ERRD	ERRD messages (ERRD$)
SSC$SYNC	Event synchronizers (SYNC$)
SSC$TIMERS	Timers (TIMER$)
SSC$ISC	InterServer Communications (ISC$)

ercode

INPUT. The error code returned by the subroutine that reports the error. If *ercode* is E$OK or invalid, ER$TEXT returns a null string.

ertext

OUTPUT. The text of the error message for subsystem *sscode* that corresponds to the error code *ercode*.

Discussion

ER$TEXT finds an error message in a message file in the SYSOVL directory or the PRIMOS internal message table, and returns the message to a variable. ER$TEXT returns the error message in the same form that ER$PRINT prints it on the terminal. ER$TEXT is similar in function to ERTXT$, except that

ER$TEXT can return error messages for particular PRIMOS subsystems. (The obsolete subroutine ERTXT$ is described in Appendix D.)

To find an error message, ER$TEXT first looks in the SYSOVL directory for the SYSOVL message file, as specified by the subsystem name appended with _ERROR_TABLE. For example, the ISC message file is named ISC$_ERROR_TABLE. Note that by convention, names of PRIMOS subsystems end with a dollar sign ($).

If the SYSOVL file exists, ER$TEXT returns the message in the SYSOVL file that corresponds to the error code in *ercode*. If the SYSOVL file does not exist, ER$TEXT looks for the message in the PRIMOS internal message table. The messages in the PRIMOS internal message table are in English. If ER$TEXT cannot find the message in the PRIMOS internal message table, it returns the values *sscode* and *ercode*.

Programs that call ER$TEXT must include the file SYSCOM>ERRORMSGHDLR.INS.*language*, where *language* is a suffix specifying the program's language. Programs should use the key values defined in this file rather than the numeric values or strings to which the key values correspond.

Loading and Linking Information

The dynamic link for ER$TEXT is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

GINFO

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:

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:

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.

GSNAM$

This subroutine returns the current PRIMOS system name.

Usage

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

CALL GSNAM$ (system_name);
```

Parameters

system_name
OUTPUT. Name of the caller's system.

Discussion

GSNAM$ can be used by any program to determine the name of the system it is running on. System names are currently limited to 6 characters in length, but Prime reserves the right to increase this limit to no more than 32 characters. Programs should account for this possibility.

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

LOV$SW

The LOV$SW function indicates whether the login-over-login function is currently permitted.

Usage

DCL LOV$SW ENTRY RETURNS (BIT (1) ALIGNED);

flag **= LOV$SW( );**

Parameters

flag

RETURNED VALUE. Returns '1'b (true) if login-over-login is not permitted, or '0'b (false) if it is permitted.

Discussion

The LOV$SW function sets the first bit of *flag* to 1 (true) if login-over-login is not permitted, or to 0 (false) if it is permitted. The login-over-login function logs a user off the system if the user executes the LOGIN command while already logged in. Because LOV$SW has no arguments, it cannot be directly called from FTN.

Example

The following fragment of C code invokes the LOV$SW function to determine whether the login-over-login function is permitted.

```
short flag, lov$sw();
flag = lov$sw();
printf("login-over-login is %sabled\n",
      (flag<0)?"en":"dis");
```

The code also displays one of the messages below:

```
Login-over-login is enabled

Login-over-login 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.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

PRI$RV

This routine 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$

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.

SNCHK$

This routine checks the validity of the system name passed to it.

Usage

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

name_ok = SNCHK$ (key, system_name);
```

Parameters

key

INPUT. Standard PRIMOS key value that defines restrictions on the parameter *system_name*. Values for keys can be added together. Possible values are

K$UPRC	Mask name to uppercase before checking.
K$NULL	Allow a null name.

system_name

INPUT/OUTPUT. System name being tested (input only, unless K$UPRC is used; in that case, input/output).

name_ok

RETURNED VALUE. Set to true ('1'b) if the system name is valid, given the restrictions of the keys; otherwise, set to false (0).

Discussion

SNCHK$ enables subsystems that deal with system names at a command interface to check the names for validity without knowing the syntax rules for system names.

Loading and Linking Information

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

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

R-mode: No special action.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

USER$

This routine 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$	Check process has given amount of time slice left.
CHG$PW	Change login validation password.
COM$AB	Expand a line using abbreviations preprocessor.
GEN$PW	Generate a new login validation password.
IDCHK$	Validate a name.
IN$LO	Determine whether a forced logout is in progress.
LOGO$$	Log out a user.
LUDEV$	Return a list of devices that a user can access.
PRJID$	Return the user's project identifier.
PTIME$	Return amount of CPU time used since login.
PWCHK$	Validate syntax of a password.
READY$	Display PRIMOS command prompt.
SID$GT	Return user number of initiating process.
SUSR$	Test whether current user is supervisor.
TI$MSG	Display standard message showing times used.
TIMDAT	Return timing information and user identification.
TMR$GINF	Return permanent time information.
TMR$GTIM	Return current system time.
TMR$LOCALCONVERT	Convert local time to Universal Time.
TMR$UNIVCONVERT	Convert Universal Time to local time.
UNO$GT	List users with same name as caller.
UTYPE$	Return user type of current process.
VALID$	Validate a name against composite identification.

ASSUR$

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 *Operator'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

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 *PRIMOS 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

E$OK	No error.
E$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$GPON	The new password is not set because generated passwords are enabled. GEN$PW generates a new login validation password for the user.
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.

If the System Administrator has enabled generated passwords for all users, CHG$PW does not set a new password for a user. In this case, the user must use

GEN$PW to generate a new password. The System Administrator may choose to disallow null passwords.

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.

Effective for PRIMOS Rev. 19.0 and subsequent revisions.

COM$AB

This routine 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

E$OK	No error.
E$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 *PRIMOS 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.

GEN$PW

This routine generates a new login validation password.

Usage

DCL GEN$PW (CHAR(16) VAR, CHAR(16) VAR, FIXED BIN(15));

CALL GEN$PW (*old_pw, new_pw, code*);

Parameters

old_pw
INPUT. The user's current password.

new_pw
OUTPUT. The new password that PRIMOS generated.

code
OUTPUT. The status code. Possible values are

E$OK	No error
E$BPAS	The *old_pw* value does not match the actual password.
E$BPAR	The *old_pw* value is in an illegal format.
E$NGPW	Computer-generated passwords are not enabled.

Discussion

GEN$PW generates a new login validation password for the user. GEN$PW can be used only when computer-generated passwords are enabled.

Loading and Linking Information

The dynamic link for GEN$PW is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

IDCHK$

This function checks that the name passed is a legal user or project name.

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 *PRIMOS User's Guide*.)
K$NULL	Allow null *id*s.
K$GRP	Check for group name.

id

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

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

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$$

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

–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 greater than 0.

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

E$OK	No error.
E$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.

Discussion

Key option 0 (Log out self) does not close MIDASPLUS ™ files or release MIDASPLUS records. MIDASPLUS users should unlock records and close files before calling LOGO$$. Refer to the *MIDASPLUS 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.

LUDEV$

This subroutine returns a list containing devices that a user can access.

Usage

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

CALL LUDEV$ (*user_no, strucptr, max_devices, code*);

Parameters

user_no

INPUT. User number of the person for whom device access information was requested. If *user_no* is 0, this indicates the current user.

strucptr

INPUT –> OUTPUT. Pointer to an area of memory that will contain the list of devices that the user can access. The Structure Description section includes the format of this structure.

max_devices

INPUT. Maximum number of devices that the caller's structure can hold.

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BPAR	Invalid parameter. Returned if *user_no* is greater than the number of users configured for the system or less than 0, or if *max_devices* is less than 1.
E$BVER	Invalid version number.
E$BFTS	*max_devices* is not large enough to hold all accessible devices for this user.

Structure Description

The parameter *strucptr* points to a structure, *device_table*, which has the following format:

```
DCL 1  device_table,
       2  version FIXED BIN,
       2  device_count FIXED BIN,
       2  device (*) CHAR(32) VAR;
```

version

INPUT. The version number of the structure to be returned. Must be set to 1.

device_count

OUTPUT. The number of devices that the specified user can access.

device

OUTPUT. Array of devices that the specified user can access.

Discussion

The devices listed are those that were specified with the ASSIGN command. Refer to the *PRIMOS Commands Reference Guide* for more information about this command.

Example

The following F77 program displays the names of up to five devices that the user can access.

```
$INSERT SYSCOM>ERRD.INS.FTN

      INTEGER*2 STRUC(87),CODE

C  The next four declarations define fields that will
C  redefine the contents of the output structure.
C  Note that elements of the device-name field are
C  34 bytes apart - CHAR(32)VAR means 32
C  bytes, plus one halfword for the count.

      INTEGER*2 DEVCT,LEN(17,5)
      CHARACTER*34 NAME(5)
      EQUIVALENCE (STRUC(2),DEVCT),(STRUC(3),LEN(1,1))
      EQUIVALENCE (STRUC(4),NAME(1))
```

```
C  Code starts here

      STRUC(1)=1
      CALL LUDEV$ (INTS(0),LOC(STRUC),INTS(5),CODE)

C  Comment if more than 5 devices

      IF (CODE.EQ.E$BFTS) PRINT *, 'More than 5 devices
                                    assigned.'
      IF (CODE.NE.E$BFTS) PRINT 100, DEVCT
100   FORMAT(' ', I1, ' devices assigned.')
      IF (DEVCT.EQ.0) CALL EXIT

C  Display all the names

      DO 10 I = 1, MIN(DEVCT, 5)
10    PRINT *, NAME(I)(1:LEN(1,I))
      CALL EXIT

      END
```

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

PRJID$

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$

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$

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 cannot 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: Not available.

READY$

READY$ displays the PRIMOS command-level prompts.

Usage

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

CALL READY$ (*format, typecode*);

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.

typecode

INPUT. Prompt type 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 *PRIMOS 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

This routine 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
OUTPUT. 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$

SUSR$ determines whether the currently executing process is the supervisor process.

Usage

```
DCL SUSR$ ENTRY RETURNS (BIT(1) ALIGNED);

susr_flag = SUSR$ ( );
```

Parameters

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

Discussion

SUSR$ determines whether the currently executing process is the supervisor process (normally User 1). The supervisor process is the process that runs at the operator console.

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

This routine 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

This routine 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.

TMR$GINF
TMR$IF

This routine returns permanent time information.

Usage

```
DCL TMR$GINF ENTRY (1, 2 FIXED BIN(31), 2 FIXED BIN(15),
                    2, 3 FIXED BIN(31), 3 FIXED BIN(31),
                    2, 3 FIXED BIN(31), 3 FIXED BIN(31),
                     2 FIXED BIN(31));

CALL TMR$GINF (TimeInfo);
```

Parameters

TimeInfo

OUTPUT. A record to which TMR$GINF returns permanent time information.

Discussion

TMR$GINF returns system time information that is not often changed. This information is known as **permanent time information.** It consists of the following items:

- The time zone in which this processor resides, expressed in milliseconds. The time zone ranges from 11 hours behind (–39600000) to 13 hours ahead (46800000) of Universal Time. Universal Time is the elapsed time since midnight of January 1, 1901, and is expressed in the mean solar time of the meridian of Greenwich.
- A value indicating whether or not daylight savings time will be or is in effect this year. The value 1 (= TRUE) indicates that daylight savings time will be or is in effect this year. The value 0 (= FALSE) indicates that daylight savings time is not and will not be in effect this year.
- The date and time of day when local time will be offset from standard local time to indicate the start of daylight savings time. This value is given in Universal Time.
- The date and time of day when local time will be reset to standard local time to indicate the end of daylight savings time. The ending time for

daylight savings time is given in Universal Time. The ending time for daylight savings time must be later than the starting time.

- The offset from standard local time that is in effect during the period from start date to end date. The offset can be negative or positive, to set time backward or forward. The offset is expressed in milliseconds, and can range from 4.66 hours behind (–16777216) to 4.66 hours ahead (16777216).

Example

The following programming example illustrates the use of TMR$GINF to return permanent time information.

```
dcl 1 PermTimeInfo BASED,
      2 CurrentTimeZone fixed bin (31),
                        /* Time zone of processor */
      2 OffsetEnable    fixed bin (15),  /* DST on */
      2 StartDate,                 /* Start date, DST */
        3 High          fixed bin (31),
        3 Low           fixed bin (31),
      2 EndDate,                   /* End date, DST */
        3 High          fixed bin (31),
        3 Low           fixed bin (31),
      2 CurrentOffset   fixed bin (31);
                           /* Offset from local time */

dcl 1 TimeInfo like PermTimeInfo;
dcl tmr$ginf entry(1, 2 fixed bin (31), 2 fixed bin (15),
                   2, 3 fixed bin (31), 3 fixed bin (31),
                   2, 3 fixed bin (31), 3 fixed bin (31),
                   2 fixed bin (31));

call tmr$ginf(TimeInfo);
```

Loading and Linking Information

The dynamic link for TMR$GINF is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

TMR$GTIM
TMR$TM

This routine returns the current system time.

Usage

```
DCL TMR$GTIM ENTRY (1, 2 FIXED BIN(31), 2 FIXED BIN(31));

CALL TMR$GTIM (CurrentTime);
```

Parameters

CurrentTime
OUTPUT. The current system time.

Discussion

TMR$GTIM returns the current system time, expressed in milliseconds. The current system time is given in Universal Time. Universal Time is the elapsed time since midnight of January 1, 1901, and is expressed in the mean solar time of the meridian of Greenwich.

Example

See the example for TMR$UNIVCONVERT.

Loading and Linking Information

The dynamic link for TMR$GTIM is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

TMR$LOCALCONVERT
TMR$LU

This routine converts local time to Universal Time.

Usage

```
DCL TMR$LOCALCONVERT ENTRY (1, 2 FIXED BIN(15),
                            2 FIXED BIN(15),
                            2 FIXED BIN(15),
                            2 FIXED BIN(15),
                           2 FIXED BIN(15),
                           1, 2 FIXED BIN(31),
                           2 FIXED BIN(31));

CALL TMR$LOCALCONVERT (LocalTime, UnivTime);
```

Parameters

LocalTime

INPUT. The local time value that is to be converted to Universal Time. Local time is expressed in a record of the following form:

```
Month: [1..12]
Day: [1..31];
Year: [0..99];
Hour: [0..23];
Minute: [0..59];
```

UnivTime

OUTPUT. The Universal Time equivalent, in milliseconds, of the local time specified by the argument *LocalTime*.

Discussion

TMR$LOCALCONVERT converts local time to Universal Time. Universal Time is the elapsed time since midnight of January 1, 1901, and is expressed in the mean solar time of the meridian of Greenwich.

The output of TMR$LOCALCONVERT can be used as input to the subroutine TMR$SABS to set an absolute timer. For information about timer subroutines, see the *Subroutines Reference V: Event Synchronization.*

Example

The following programming example calls TMR$LOCALCONVERT to convert the local time of 12 noon, July 4, 1987 to Universal Time.

```
dcl 1 LocTime based,
      2 Month      fixed bin,
      2 Day        fixed bin,
      2 Year       fixed bin,
      2 Hour       fixed bin,
      2 Minute     fixed bin;

dcl 1 AbsoluteTime based,
      2 High  fixed bin (31),
      2 Low   fixed bin (31);

dcl 1 LocalTime like LocTime;
dcl 1 UnivTime like AbsoluteTime;
dcl tmr$localconvert external entry
                  (1, 2 fixed bin, 2 fixed bin,
                   2 fixed bin, 2 fixed bin,
                   2 fixed bin,
                   1, 2 fixed bin (31),
                   2 fixed bin(31));

LocalTime.Month = 7;
LocalTime.Day = 4;
LocalTime.Year = 87;
LocalTime.Hour = 12;
LocalTime.Minute = 0;
call tmr$localconvert(LocalTime,UnivTime);
```

Loading and Linking Information

The dynamic link for TMR$LOCALCONVERT is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

TMR$UNIVCONVERT
TMR$UL

This routine converts Universal Time to local time.

Usage

```
DCL TMR$UNIVCONVERT ENTRY (1, 2 FIXED BIN(31),
                           2 FIXED BIN(31), 1, 2 FIXED BIN(15),
                           2 FIXED BIN(15), 2 FIXED BIN(15),
                           2 FIXED BIN(15), 2 FIXED BIN(15));

CALL TMR$UNIVCONVERT (UnivTime, LocalTime);
```

Parameters

UnivTime

INPUT. The Universal Time value that TMR$UNIVCONVERT is to convert to local time, expressed in milliseconds.

LocalTime

OUTPUT. The local time equivalent of the Universal Time specified by the argument *UnivTime*. TMR$UNIVCONVERT returns this information to a record of the following form:

```
Month: [1..12];
Day: [1..31];
Year: [0..99];
Hour: [0..23];
Minute: [0..59];
```

Discussion

TMR$UNIVCONVERT converts Universal Time to local time. Universal Time is the elapsed time since midnight of January 1, 1901, and is expressed in the mean solar time of the meridian of Greenwich.

As input, TMR$UNIVCONVERT can use output from the subroutine TMR$GTIM.

Example

The following programming example calls TMR$GTIM to get the current system time in milliseconds, and calls TMR$UNIVCONVERT to convert the system time to Universal Time.

```
dcl 1 LocTime based,
      2 Month       fixed bin,
      2 Day         fixed bin,
      2 Year        fixed bin,
      2 Hour        fixed bin,
      2 Minute      fixed bin;

dcl 1 AbsoluteTime based,
      2 High  fixed bin (31),
      2 Low   fixed bin (31);

dcl 1 LocalTime like LocTime;
dcl 1 UnivTime like AbsoluteTime;

dcl tmr$gtim external entry (1, 2 fixed bin (31),
                             2 fixed bin (31));
dcl tmr$univconvert external entry (1, 2 fixed bin(31),
                                       2 fixed bin(31),
                                    1, 2 fixed bin,
                                       2 fixed bin,
                                       2 fixed bin,
                                       2 fixed bin,
                                       2 fixed bin);

/* get current time in milliseconds */
call tmr$gtim(UnivTime);

/* convert it into local time */
call tmr$univconvert(UnivTime,LocalTime);
```

Loading and Linking Information

The dynamic link for TMR$UNIVCONVERT is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

UNO$GT

This routine 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$

This routine 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 *Subroutines Reference I: Using Subroutines*. 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: Not available.

VALID$

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

E$OK	No error.
E$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.

System Status and Metering Routines

This section describes the following subroutines:

Routine	*Function*
DS$AVL	Return data about a disk partition.
DS$ENV	Return data about a process's environment.
DS$UNI	Return data about file units.
G$METR	Return a variety of metering information.

DS$AVL

This subroutine returns information about a disk partition in a structure.

Usage

DCL DS$AVL ENTRY (POINTER, FIXED BIN, FIXED BIN);

CALL DS$AVL (*struc_ptr, disk_no, code*);

Parameters

struc_ptr

INPUT –> OUTPUT. A pointer to a structure that will contain the output information. See below.

disk_no

INPUT. The logical disk partition number (ldev) for which information is being requested.

code

OUTPUT. Standard error code. Possible values include

E$OK	No error.
E$BVER	*version* is an illegal value.
E$BPAR	Invalid logical disk number specified.
E$NINF	Specified partition does not exist or is not added.
E$FNTF	The area of the disk that contains the availability information cannot be found or accessed.
E$IREM	The specified partition is mounted on a remote node.

Structure Description

The parameter *struc_ptr* points to a structure, *avail_list*, of the following format:

```
DCL 1  avail_list,
       2  version FIXED BIN,
       2  disk_name CHAR(32)VAR,
       2  partition_size FIXED BIN(31),
       2  available_size FIXED BIN(31),
       2  date_time_saved FIXED BIN(31);
```

version

INPUT. The version number of the structure to be returned. Must be set to 1.

disk_name

OUTPUT. The name of the partition.

partition_size

OUTPUT. The maximum capacity of the partition, in records.

available_size

OUTPUT. The number of free records in the partition.

date_time_saved

OUTPUT. The date and time when the partition was last backed up with the MAGSAV utility, in file–system date format. If the partition has never been backed up with MAGSAV, this field is zero. File–system date format is described in Appendix C.

Discussion

DS$AVL can only return information about disks listed in your system's disk table: local disks and specifically configured remote disks. The logical device (ldev) numbers for these disk partitions are shown in the STATUS DISKS display. These ldev numbers are returned by the routine LUDSK$ and used by the routine AT$LDEV.

If you have the common file system name space (i.e., the Name Server) configured on your system, other disks may be available for user access. To list all of the disks common to your name space, use the LIST_MOUNTS command or the NAM$L_GMT subroutine. DS$AVL cannot return information about these additional disks.

Loading and Linking Information

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

V-mode and I-mode with unshared libraries: Use the DYNT subcommand of BIND to create a dynamic link to DS$AVL. For information about BIND, see *Programmer's Guide to BIND and EPFs.*

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

DS$ENV

This subroutine returns information about the user's process.

Usage

DCL DS$ENV ENTRY (FIXED BIN, POINTER, FIXED BIN);

CALL DS$ENV (*user_no, struc_ptr, code*);

Parameters

user_no

INPUT. The number of the user for which information is requested. If *user_no* is zero, the current user is assumed as the default.

struc_ptr

INPUT –> OUTPUT. A pointer to a structure that will contain the output information after the call. See the Structure Description section, below.

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BVER	*version* is an illegal value.
E$BPAR	An illegal value was specified for *user_no*.
E$NRIT	Caller may not access information about another user. See Discussion, below.

Structure Description

The parameter *struc_ptr* points to the structure *env_list*, shown below:

```
DCL 1  env_list,
       2  version FIXED BIN,
       2  abbrev_fname CHAR(80)VAR,
       2  como_sw BIT ALIGNED,
       2  comi_sw BIT ALIGNED,
       2  comi_unit FIXED BIN,
       2  command_level FIXED BIN,
       2  erase_char CHAR,
       2  kill_char CHAR,
       2  default_uts FIXED BIN,
       2  current_uts FIXED BIN,
       2  auto_log_clock FIXED BIN,
       2  cpu_limit FIXED BIN(31),
       2  login_limit FIXED BIN(31),
       2  quit_inhibits FIXED BIN,
       2  group_count FIXED BIN,
       2  group_names (32)CHAR(32)VAR,
       2  rid_count FIXED BIN,
       2  rid_info(16),
          3  remote_node_name CHAR(32)VAR,
          3  remote_user_id CHAR(32)VAR,
          3  remote_project_id CHAR(32)VAR;
```

version

INPUT. The version number of the structure. Must be set to 1.

abbrev_fname

OUTPUT. Filename of the currently active abbreviation file. If no abbreviation file is active, a null string is returned.

como_sw

OUTPUT. If a command output file is enabled, this is set to true ('1'b).

comi_sw

OUTPUT. If a command input file is enabled, this is set to true ('1'b).

comi_unit

OUTPUT. The unit number of the current command input file. The value of this field is undefined if there is no current command input file.

command_level

OUTPUT. User's current command level.

erase_char

OUTPUT. User's current erase character.

kill_char

OUTPUT. User's current kill character.

default_uts

OUTPUT. Default user time slice in units of 1.024 milliseconds, as a negative number. Time slices are described in the *System Architecture Reference Guide.*

current_uts

OUTPUT. Current user time slice in units of 1.024 milliseconds, as a negative number.

auto_logo_clock

OUTPUT. Number of minutes remaining until the user is logged out due to inactivity.

cpu_limit

OUTPUT. CPU time remaining, in milliseconds. If the current process is a batch job and has a CPU time limit set, the value is nonzero.

login_limit

OUTPUT. Login time remaining, in minutes. This value is nonzero if the current process is a batch job, and has an elapsed time limit set. This value is also nonzero if the process is in the logout grace period (the process is processing a LOGOUT$ on-unit).

quit_inhibits

OUTPUT. The QUIT inhibit count. This is equivalent to the number of times that BREAK$ has been called to defer recognition of terminal quits.

group_count

OUTPUT. The number of ACL groups to which the user belongs. The names of these groups are contained in *group_names.*

group_names

OUTPUT. An array containing the names of the ACL groups to which the user belongs. Only *group_count* elements of *group_names* are set.

rid_count

OUTPUT. The number of added remote IDs. These IDs are listed in *rid_info.* See the description of the ADD_REMOTE_ID command, in the *PRIMOS User's Guide,* for more information.

rid_info

OUTPUT. Each element of this array is a structure containing information about a remote ID, as described in the following fields. Only *rid_count* elements of *rid_info* are set.

rid_info.remote_node_name

OUTPUT. The node name for this remote ID.

rid_info.remote_user_id

OUTPUT. The user ID for this remote ID.

rid_info.remote_project_id

OUTPUT. The project ID for this remote ID. If no project ID has been set, this field contains the null string.

Discussion

Any process may obtain information about itself. However, only the system operator, and phantom jobs spawned by the operator, may access information about another user.

Users of PRIMOS Rev. 21.0 may need to DYNT this subroutine to BIND successfully. Users of PRIMOS Rev. 22.0 and subsequent revisions should not have this problem.

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

DS$UNI

This subroutine returns information about file units.

Usage

```
DCL DS$UNI ENTRY (FIXED BIN, FIXED BIN, FIXED BIN,
                  CHAR(128)VAR, POINTER, FIXED BIN);

CALL DS$UNI (key, user_no, unit_no, full_path, struc_ptr, code);
```

Parameters

key

INPUT. Indicates the information to be returned. Possible values are

K$UNIT	Returns information about a specific unit
K$CURA	Returns information about the current attach point
K$HOMA	Returns information about the home attach point
K$INIA	Returns information about the initial attach point
K$COMO	Returns information about the command output file unit
K$NEXT	Returns information about the next open unit whose pathname contains *full_path* as a prefix

user_no

INPUT. The number of the user for which information is requested. If *user_no* is zero, the current user is assumed as the default.

unit_no

INPUT/OUTPUT. If *key* is K$UNIT, information is returned about unit *unit_no*.

If *key* is K$NEXT, information is returned about the next unit whose number is greater than *unit_no*. See Discussion, below, for a full description.

If *key* has another value, *unit_no* is ignored.

full_path

If *key* is K$NEXT, *full_path* contains the prefix on which to match. See Discussion, below, for a description.

struc_ptr

INPUT –> OUTPUT. A pointer to a structure that will contain the output information after the call. See Structure Description, below.

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BVER	*version* is an illegal value.
E$BKEY	An illegal value was specified for *key*.
E$BPAR	An illegal value was specified for *user_no*.
E$NRIT	Caller may not access information about another user.
E$BUNT	Invalid value for *unit_no*.
E$UNOP	Either the specified unit was not open or the attach point was not attached to any directory, or, if *key* is K$NEXT, no further open units were found.
E$BFTS	The pathname is longer than 128 characters and has not been returned to the field *pathname*; other fields have been set.
E$SHDN	The disk has been shut down.

Structure Description

The parameter *struc_ptr* points to a structure, *unit_list*, of the following format:

```
DCL 1  unit_list,
       2  version FIXED BIN,
       2  remote_unit BIT(1) ALIGNED,
       2  status,
          3 modified BIT,
          3 sysuse BIT,
          3 shut_down BIT,
          3 no_close BIT,
          3 disk_error BIT,
          3 file_type BIT(3),
       2  open_mode,
          3  not_used1 BIT(3),
          3  vmfa_read BIT,
          3  not_used2 BIT,
          3  attach_pt BIT,
          3  write BIT,
          3  read BIT,
       2  rwlock FIXED BIN,
```

```
2  access_bits,
   3  protect BIT,
   3  delete BIT,
   3  add BIT,
   3  list BIT,
   3  use BIT,
   3  execute BIT,
   3  write BIT,
   3  read BIT,
   3  owner BIT,
   3  not_used BIT(7),
2  position FIXED BIN (31),
2  system_name CHAR(32) VAR,
2  pathname CHAR(128) VAR;
```

version

INPUT. The version number of the structure. Must be set to 1.

remote_unit

OUTPUT. If the unit is open on another node, this is set to '1'b.

status

OUTPUT. These 8 bits indicate the file's status, as described in the following fields. These fields are valid only if the file is open on the local system (*remote_unit* is '0'b).

status.modified

OUTPUT. If the file has been modified, this bit is set.

status.sysuse

OUTPUT. If the file is open for system use, this bit is set.

status.shut_down

OUTPUT. If the file's disk has been shut down, this bit is set.

status.no_close

OUTPUT. Some open file units may not be closed. Attempts to close them produce an error code. If the unit may not be closed, this bit is set.

status.disk_error

OUTPUT. This bit is set if there has been a disk error on this file.

status.file_type

OUTPUT. This three-bit field holds a number between 0 and 7, indicating the file type. Types are defined with the specification of SRCH$$.

open_mode

OUTPUT. These bits indicate access for the open file, as described in the following fields. These fields are valid only if the file is open on the local system (*remote_unit* is '0'b).

open_mode.vmfa_read

OUTPUT. If the file is open for VMFA read (has been opened with the K$VMR key), this bit is set.

open_mode.attach_pt

OUTPUT. If the unit is an attach point (initial, home, or current), this bit is set.

open_mode.write

OUTPUT. If writing is permitted, this bit is set.

open_mode.read

OUTPUT. If reading is permitted, this bit is set.

rwlock

OUTPUT. This field indicates the allowed concurrent access to the file or directory. This field is valid only if the file is open on the local system (*remote_unit* is '0'b). Possible values are

K$DFLT	Allow one reader or one writer.
K$EXCL	Allow any number of readers, or one writer.
K$UPDT	Allow any number of readers and one writer.
K$NONE	Allow any number of readers and writers.

access_bits

OUTPUT. These bits indicate the access the user has to the file or directory, as described in the following fields. These access flags are those defined by the ACL subsystem. These fields are valid only if the file is open on the local system (*remote_unit* is '0'b).

The bits indicate Owner, Protect, Delete, Add, List, Use, Execute, Write, and Read access.

position

OUTPUT. The file position. If the file is open for VMFA access, *position* is always zero. This field is valid only if the file is open on the local system (*remote_unit* is '0'b).

system_name

OUTPUT. Gives the name of the system on which the remote unit is open. If the file is open on the local system, a null string is returned.

pathname

OUTPUT. If the file is open on a local disk partition, this is the file's pathname. If the file is open on a remote disk partition, this field returns only the top-level partition name. The open file may be on that partition, or may be on a lower-level partition under the top-level partition. If the file is in a root-directed portal, no partition name is returned.

Discussion

DS$UNI returns information about the user's file units. If the Name Server is in use, this subroutine can return file unit information from any disk common to the user's name space. DS$UNI returns the same information from remote disks configured on your system and remote disks accessed through the Name Server.

If *key* is K$NEXT, information is returned for the next locally opened unit greater than *unit_no* whose pathname contains the string *full_path* (which may be null) as a prefix. On a successful return, *unit_no* is updated to indicate the unit number for which information is being returned. To scan all the user's units, the programmer should set *unit_no* initially to –1 and call DS$UNI, with *key* K$NEXT, in a loop that terminates on return of error code E$UNOP.

This routine includes the function of the existing routines FINFO$ and GPATH$. However, FINFO$ and GPATH$ will continue to be fully supported.

Only the system operator, and phantom jobs spawned by the operator, may access information about another user.

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

G$METR

G$METR returns system metering information, such as that returned by the USAGE command.

Usage

```
DCL G$METR ENTRY (FIXED BIN(15), PTR OPTIONS(SHORT),
                  FIXED BIN(15), FIXED BIN(15),
                  FIXED BIN(15), FIXED BIN(15));

CALL G$METR (meter_key+user_key, bufptr, bufsize, user_no, version,
             code);
```

Parameters

meter_key

INPUT. A code that specifies the type of metering information to be returned. Possible values are

1	General system meters
2	File system meters
3	Interrupt process meters
4	System meters for an individual user
5	Meters for memory use
6	Meters for disk use
7	Meters for ROAM use
8	Scheduler information meters

For Rev. 23.0 and later revisions, if you specify a *meter_key* value of 4, you may also specify a *user_key* value. If you specify any other *meter_key* value, the *user_key* value is ignored. Using the *user_key* subparameter and the *user_no* parameter, you can define a sequential search of users. *user_no* specifies the starting user number and *user_key* specifies the type of users to search.

The *user_key* values are as follows:

0	No user searching; specified user only
1024	Searches for the next existing user (of any type)
2048	Searches for the next existing terminal-user type
3072	Searches for the next existing phantom-user type
4096	Searches for the next existing kernel-user type

Specify a *meter_key* of 4, a plus sign, and one of these *user_key* values. Do not include blanks or more than one *user_key* value. If you do not specify a key value, G$METR works exactly as it did prior to Rev. 23.0: it locates the user number specified in *user_no*.

bufptr

INPUT –> OUTPUT. Pointer to an area in memory that will contain metering information for the *meter_key* chosen. The Structure Description section includes the format of this structure.

bufsize

INPUT. Maximum size of the caller's buffer.

user_no

INPUT. User number. This value is used only if the value of *meter_key* is 4. You can either use this field to specify an individual user by user number or to initiate a sequential search for a user number.

To request a specific user number, set *meter_key* to 4. Do not specify a *user_key* subparameter. Specify the user number in this parameter. You can specify the current user by setting *user_no* to 0.

To request a search for a user number, you must be running PRIMOS Rev. 23.0. Set *meter_key* to 4 and specify a *user_key* subparameter. Specify the starting user number for a sequential search in this parameter. G$METR returns information on the first user number equal to or greater than the number you specified that satisfies the search criteria.

version

INPUT. The version number for the structure pointed to by *bufptr*. Specify the version number that corresponds to the revision of PRIMOS that you are using. Possible values are

5	Rev. 21.0
6	Rev. 22.0
7	Rev. 22.1
8	Rev. 23.0

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BPAR	Invalid parameter. Occurs when an invalid *meter_key*, *user_key*, *user_no*, or *version* is supplied. A *user_no* is invalid if it is not within the range of existing user numbers.
E$BFTS	*bufsize* is not large enough to hold the information requested.
E$UINF	Search for user number unsuccessful. There is no existing user with a user number equal to or greater than *user_no* that satisfies the user type restriction specified by *user_key*.

Structure Description

The parameter *bufptr* points to one of the structures shown below, depending on the *meter_key* specified.

General System Meters: When *meter_key* is 1, the parameter *bufptr* points to the structure *meter_sys*. Specify 13 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_sys,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  cptot FIXED BIN(31),
       2  iotot FIXED BIN(31),
       2  gclock FIXED BIN(31),
       2  iocnt FIXED BIN(31),
       2  nusr FIXED BIN(15),
       2  cptick FIXED BIN(15),
       2  clrate FIXED BIN(15);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

unused_field

OUTPUT. Reserved.

cptot

OUTPUT. CPU time used, in CPU ticks, since system boot. The length of a CPU tick is returned in the *cptick* field of this structure. (Note that a CPU tick is *not* the same as a real time clock tick.) To compute the CPU time used in milliseconds, multiply the value of *cptot* by the value of *cptick*, then divide by 1000.

iotot

OUTPUT. I/O time used, in real time clock ticks, since system boot. The length of a real time clock tick is returned in the *clrate* field of this structure. (Note that a real time clock tick is *not* the same as a CPU tick.) To compute the I/O time used in milliseconds, multiply the value of *iotot* by 1000, then divide by the value of *clrate*.

gclock

OUTPUT. Elapsed time, in real time clock ticks, since system boot. The length of a real time clock tick is returned in the *clrate* field of this structure. (Note that a real time clock tick is *not* the same as a CPU tick.) To compute this elapsed time used in milliseconds, multiply the value of *gclock* by 1000, then divide by the value of *clrate*.

iocnt

OUTPUT. Number of disk I/O operations performed since system boot.

nusr

OUTPUT. Number of users configured on the system.

cptick

OUTPUT. Length of a CPU tick. *cptick* is a time measurement, expressed in microseconds per CPU tick. The standard value is 1024 microseconds per CPU tick. (Note that a CPU tick is *not* the same as a real time clock tick.)

clrate

OUTPUT. Length of a real time clock tick, expressed in real time clock ticks per second. The value of this field is CPU-dependent. (Note that a real time clock tick is *not* the same as a CPU tick.)

File System Meters: When *meter_key* is 2, the parameter *bufptr* points to the structure *meter_fs*. Specify 18 for *bufsize* and declare the following structure in your program:

```
DCL 1 meter_fs,
      2  version FIXED BIN(15),
      2  unused_field FIXED BIN(15),
      2  pg_faults FIXED BIN(31),
      2  locate,
         3  misses FIXED BIN(31),
         3  found FIXED BIN(31),
         3  same FIXED BIN(31),
         3  used FIXED BIN(31),
      2  blkcnt,
         3  read_cnt FIXED BIN(31),
         3  write_cnt FIXED BIN(31),
         3  awrite_cnt FIXED BIN(31);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

unused_field

OUTPUT. Reserved.

pg_faults

OUTPUT. Number of page faults since system boot.

locate

OUTPUT. A group of fields containing information about locate meters.

locate.misses

OUTPUT. Number of times since system boot that any process needed a disk record that was not in a Locate buffer.

locate.found

OUTPUT. Number of times since system boot that a disk record needed was in a Locate buffer.

locate.same

OUTPUT. Number of times since system boot that a process needed a physical record that it already owned.

locate.used

OUTPUT. Number of Locate buffers in use.

blkcnt

OUTPUT. A group of fields containing read/write information about ROAM direct I/O since system boot.

blkcnt.read_cnt

OUTPUT. Number of read operations performed.

blkcnt.write_cnt

OUTPUT. Number of write operations performed.

blkcnt.awrite_cnt

OUTPUT. Number of asynchronous write operations performed.

Interrupt Process Meters: If *meter_key* is 3, the parameter *bufptr* points to the structure *meter_int*. G$METR uses the version of *meter_int* that corresponds to the *version* input parameter value. At Rev. 21.0, specify 62 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_int,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  cpused(30) FIXED BIN(31);
```

At Rev. 22.0 and subsequent revisions, specify 64 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_int,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  cpused(31) FIXED BIN(31);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

unused_field

OUTPUT. Reserved.

cpused

OUTPUT. An array giving the amount of CPU time used since system boot by each interrupt process, such as the clock process, the AMLC driver process,

and the disk processes, as shown below. Rev. 22.0 adds entry 31 (total CPU time used by other interrupt processes) to the *cpused* array.

Index in the cpused Array	*Description*
1	Debugger process
2	Backstop process
3	Second backstop process
4	Clock process
5	Front stop process
6	SMLC process
7	AMLC process
8	MPC process
9	MPC process
10	MPC4/Versatec process
11	MPC4/Versatec process
12	MPC4/Versatec process
13	RINGNET™ process
14	RINGNET process
15	First disk process
16	Second disk process
17	Third disk process
18	Fourth disk process
19	Fifth disk process
20	Sixth disk process
21	Seventh disk process
22	Eighth disk process
23	AMLC process
24	SMLC process
25	IPQ process
26	IPQ process

Index in the cpused Array	*Description*
27	Maintenance Processor process
28	AMLC process
29	AMLC process
30	NTS process
31	Rev. 22.0 and subsequent revisions: Total CPU time used by other interrupt processes (usually 0)

Per User System Meters: When *meter_key* is 4, the parameter *bufptr* points to the structure *meter_user*.

For revisions earlier than Rev. 22.1, specify 43 for *bufsize*, specify the appropriate value for *version*, and declare the following structure in your program:

```
DCL 1  meter_user,
       2  version FIXED BIN(15),
       2  user_type FIXED BIN(15),
       2  log_date FIXED BIN(15),
       2  log_time FIXED BIN(15),
       2  log_nam CHAR(32),
       2  projid CHAR(32),
       2  cpu_used FIXED BIN(31),
       2  io_used FIXED BIN(31),
       2  unused_field1 FIXED BIN(31),
       2  unused_field2 FIXED BIN(15);
```

At Rev. 22.1, specify 67 for *bufsize*, specify 7 for *version*, and declare the following structure in your program:

```
DCL 1  meter_user,
       2  version FIXED BIN(15),
       2  user_type FIXED BIN(15),
       2  log_date FIXED BIN(15),
       2  log_time FIXED BIN(15),
       2  log_nam CHAR(32),
       2  projid CHAR(32),
       2  cpu_used FIXED BIN(31),
       2  io_used FIXED BIN(31),
       2  unused_field1 FIXED BIN(31),
       2  unused_field2 FIXED BIN(15),
       2  rdcn FIXED BIN(31),
       2  unused_field3  FIXED BIN(31),
       2  wrcn  FIXED BIN(31),
```

```
        2  reserved1  FIXED BIN(31),
        2  reserved2  FIXED BIN(31),
        2  unused_field4  FIXED BIN(31),
        2  reserved3  FIXED BIN(31),
        2  reserved4 FIXED BIN(31),
        2  lord  FIXED BIN(31),
        2  lowrpf  FIXED BIN(31),
        2  lowrca  FIXED BIN(31),
        2  lowpffor  FIXED BIN(31);
```

At Rev. 23.0 and subsequent revisions, specify 70 for *bufsize*, specify 8 for *version*, and declare the following structure in your program:

```
DCL 1  meter_user,
       2  version FIXED BIN(15),
       2  user_type FIXED BIN(15),
       2  log_date FIXED BIN(15),
       2  log_time FIXED BIN(15),
       2  log_nam CHAR(32),
       2  projid CHAR(32),
       2  cpu_used FIXED BIN(31),
       2  io_used FIXED BIN(31),
       2  unused_field1 FIXED BIN(31),
       2  unused_field2 FIXED BIN(15),
       2  rdcn  FIXED BIN(31),
       2  unused_field3  FIXED BIN(31),
       2  wrcn  FIXED BIN(31),
       2  reserved1  FIXED BIN(31),
       2  reserved2  FIXED BIN(31),
       2  unused_field4  FIXED BIN(31),
       2  reserved3  FIXED BIN(31),
       2  reserved4 FIXED BIN(31),
       2  lord  FIXED BIN(31),
       2  lowrpf  FIXED BIN(31),
       2  lowrca  FIXED BIN(31),
       2  lowpffor  FIXED BIN(31),
       2  user_num  FIXED BIN(15),
       2  user_pri  FIXED BIN(15),
       2  majorts  FIXED BIN(15);
```

The contents of the fields of the *meter_user* structure are described below.

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

user_type

OUTPUT. Type of user. The UTYPE$ description in Chapter 2 of this volume contains a list of possible user types. For local users, the structure contains the information below; for remote users, only the *projid* field is significant.

log_date

OUTPUT. Date of login, in file-system date format. This field corresponds to the *year*, *month*, and *day* fields of the file-system date format structure described in Appendix C. When combined, the *log_date* and *log_time* fields comprise a complete file-system date format, as described in Appendix C.

log_time

OUTPUT. Time of login, expressed in total quadseconds since midnight. This field corresponds to the *quadseconds* field of the file-system date format structure described in Appendix C.

log_nam

OUTPUT. Login ID in the process descriptor block.

projid

OUTPUT. User's project ID; this contains the string REMOTE if the user is logged through this system to get to another system, or is a remote user on this system.

cpu_used

OUTPUT. Amount of CPU time used since login, expressed in units of 1.024 milliseconds (= 1,024 microseconds).

io_used

OUTPUT. Amount of disk I/O time used since login, expressed in units of real time clock ticks. The clock tick rate for the given processor is returned by G$METR in the *clrate* entry of the General System Meter structure. (See above.)

unused_field1

OUTPUT. Reserved for future use. Currently always returns a value of zero.

unused_field2

OUTPUT. Reserved for future use. Currently always returns a value of zero.

rdcn

OUTPUT. The number of synchronous reads this process has performed since login. This field is used only for Rev. 22.1 and subsequent revisions.

unused_field3

OUTPUT. Reserved for future use. Currently always returns a value of zero.

wrcn

OUTPUT. The number of synchronous writes this process has performed since login. This field is used only for Rev. 22.1 and subsequent revisions.

reserved1

OUTPUT. Reserved for internal use.

reserved2

OUTPUT. Reserved for internal use.

unused_field4

OUTPUT. Reserved for future use. Currently always returns a value of zero.

reserved3

OUTPUT. Reserved for internal use.

reserved4

OUTPUT. Reserved for internal use.

lord

OUTPUT. The number of reads from disk to a Locate buffer that this process has performed since login. When this meter is incremented, the *rdcn* meter is also incremented. This field is used only for Rev. 22.1 and subsequent revisions.

lowrpf

OUTPUT. The number of times that this process has, since login, performed a synchronous write to disk of a Locate buffer. A user process performs a synchronous write to disk under two circumstances: 1) when it makes a read or write request for a disk record that is not currently in the Locate buffer pool when no free nonmodified buffers are available, or 2) when it force writes a Locate buffer to disk. When the first circumstance occurs, the process writes the buffer in order to clear space for its own data, and in so doing is charged with performing a write. The user process is then free to acquire this buffer. In addition, the Locate flush process writes modified buffers to disk directly. When this meter is incremented, the *wrcn* meter is also incremented. This field is used only for Rev. 22.1 and subsequent revisions.

lowrca

OUTPUT. The number of writes of Locate buffers to disk that this process has caused since login. The first process to modify a Locate buffer is charged with causing the write of that buffer to disk, even though another process such

as the Locate flush process may perform the write of that buffer to disk. This field is used only for Rev. 22.1 and subsequent revisions.

lowpffor

OUTPUT. The number of forced writes to Locate buffers done by this process. When this meter is incremented the *lowrpf* meter is also incremented. This field is used only for Rev. 22.1 and subsequent revisions.

user_num

OUTPUT. Current user number. This is the user number of the first logged-in user returned by the sequential search established by the *user_no* parameter. If the specified user number was invalid, or the search failed to locate a valid user number, G$METR sets this field to zero and returns an error code to the *code* parameter. This field is used only for Rev. 23.0 and subsequent revisions.

user_pri

OUTPUT. The user's priority. Possible values include 0 through 4, –1 (idle queue), and –2 (suspend queue). This field is used only for Rev. 23.0 and subsequent revisions.

majorts

OUTPUT. The length of the user's major time slice, in milliseconds. You can set the length of a major time slice by using the CHAP command. This field is used only for Rev. 23.0 and subsequent revisions.

Memory Meters: When *meter_key* is 5, the parameter *bufptr* points to the structure *meter_mem*. The three possible forms of the *meter_mem* structure are described below. The size of the *segments*, *physical_mem*, and *wired_mem* arrays in each structure indicates the number of users that can be configured.

At Rev. 21.0, specify 771 for *bufsize*, specify 5 for *version*, and declare the following structure in your program:

```
DCL 1  meter_mem,
       2  version FIXED BIN(15),
       2  max_segs FIXED BIN(15),
       2  total_segs FIXED BIN(15),
       2  max_pages FIXED BIN(15),
       2  total_pages FIXED BIN(15),
       2  wired_pages FIXED BIN(15),
       2  segments(255) FIXED BIN(15),
       2  physical_mem(255) FIXED BIN(15),
       2  wired_mem(255) FIXED BIN(15);
```

At Rev. 22.0, specify 4964 for *bufsize*, specify 6 for *version*, and declare the following structure in your program.

```
DCL 1  meter_mem,
       2  version FIXED BIN(15),
       2  max_segs FIXED BIN(15),
       2  total_segs FIXED BIN(15),
       2  max_pages FIXED BIN(31),
       2  total_pages FIXED BIN(31),
       2  wired_pages FIXED BIN(31),
       2  segments(991) FIXED BIN(15),
       2  physical_mem(991) FIXED BIN(31),
       2  wired_mem(991) FIXED BIN(31);
```

At Rev. 22.1 and subsequent revisions, specify 4984 for *bufsize*, specify the appropriate *version*, and declare the following structure in your program:

```
DCL 1  meter_mem BASED,
       2  version FIXED BIN(15),
       2  max_segs FIXED BIN(15),
       2  total_segs FIXED BIN(15),
       2  max_pages FIXED BIN(31),
       2  total_pages FIXED BIN(31),
       2  wired_pages FIXED BIN(31),
       2  segments(991) FIXED BIN(15),
       2  physical_mem(991) FIXED BIN(31),
       2  wired_mem(991) FIXED BIN(31),
       2  init_pgrecs(8) FIXED BIN(15),
       2  cur_pgrecs(8) FIXED BIN(15),
       2  cur_vmfa FIXED BIN(15),
       2  tot_vmfa FIXED BIN(15),
       2  iopfcn FIXED BIN(15);
```

The contents of the fields of the *meter_user* structure are described below. Use fields *init_pgrecs* through *iopfcn* only in the 22.1 structure. Use the other fields in all forms of the structure.

version

OUTPUT. Version number of the structure. Returns the value specified for the *version* input parameter (see above).

max_segs

OUTPUT. Maximum number of segments in the system.

total_segs

OUTPUT. Total segments in use.

max_pages

OUTPUT. Maximum number of physical pages in the system. Note that this field is smaller for Version 5 than it is for Version 6 and subsequent versions. (See the declarations above.) If you specify Version 5 for a system with more memory than can be expressed in 16 bits, G$METR returns a value of 0 to *max_pages*.

total_pages

OUTPUT. Total pages in use.

wired_pages

OUTPUT. Number of wired pages in the system.

segments

OUTPUT. An array, indexed by user number. Each element gives the number of segments allocated to a user. The size of this array is 255 for Version 5 and 991 for Version 6 and subsequent versions.

physical_mem

OUTPUT. An array, indexed by user number. Each element gives the number of pages allocated to each user. The size of this array is 255 for Version 5 and 991 for Version 6 and subsequent versions.

wired_mem

OUTPUT. An array, indexed by user number. Each element gives the number of wired pages allocated to each user. The size of this array is 255 for Version 5 and 991 for Version 6 and subsequent versions.

init_pgrecs

OUTPUT. The initial number of paging records available, divided by eight. This field is used only for Version 7 (Rev. 22.1) and subsequent versions.

cur_pgrecs

OUTPUT. The number of currently available paging records, divided by eight. This field is used only for Version 7 (Rev. 22.1) and subsequent versions.

cur_vmfa

OUTPUT. Number of VMFA segments currently in use. This field is used only for Version 7 (Rev. 22.1) and subsequent versions.

tot_vmfa

OUTPUT. Total number of VMFA segments configured for this system. This field is used only for Version 7 (Rev. 22.1) and subsequent versions.

iopfcn

OUTPUT. Total I/O caused by page faults. This field is used only for Version 7 (Rev. 22.1) and subsequent versions.

Note If you use a Version 5 *meter_mem* structure on a system using Rev. 22.0 or a later revision, 0 is returned to *max_pages*, *total_pages*, and *wired_pages* when the amount of memory overflows these fields (> 65535 pages). If you use a Version 5 *meter_mem* structure on a system using Rev. 22.0 or a later revision with more than 255 users configured, information on users numbered 1 through 255 only is returned to *segments*, *physical_mem*, and *wired_mem*. The number of configured users is returned to *meter_sys.nusr*.

Disk Meters: When *meter_key* is 6, the parameter *bufptr* points to the structure *meter_disk*. At Rev 20.2, specify 74 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_disk,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  q_waits FIXED BIN(31),
       2  dma_overruns FIXED BIN(31),
       2  hangs FIXED BIN(31),
       2  io_time(0:3, 0:3) FIXED BIN(31),
       2  io_cnt(0:3, 0:3) FIXED BIN(31),
       2  async_write_cnt FIXED BIN(31);
```

At Rev 21.0 and subsequent revisions, specify 266 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_disk,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  q_waits FIXED BIN(31),
       2  dma_overruns FIXED BIN(31),
       2  hangs FIXED BIN(31),
       2  io_time(0:7, 0:7) FIXED BIN(31),
       2  io_cnt(0:7, 0:7) FIXED BIN(31),
       2  async_write_cnt FIXED BIN(31);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

unused_field

OUTPUT. Reserved.

q_waits

OUTPUT. Number of times since system boot that a process had to wait for allocation of a disk request block.

dma_overruns

OUTPUT. Number of disk operations since system boot that resulted in DMA overrun errors.

hangs

OUTPUT. Number of operations since system boot that caused a disk controller to hang and time out.

io_time

OUTPUT. Amount of I/O, in seconds, that each device has used since it was initialized. This field is an eight-element, two-dimensional array. The first index is the controller number, and the second index is the device number.

io_cnt

OUTPUT. Number of I/O operations performed by each device since system boot. This field is an eight-element, two-dimensional array. The first index is the controller number, and the second index is the device number.

async_write_cnt

OUTPUT. Number of asynchronous ROAM write operations performed since ROAM initialization.

ROAM Meters: When *meter_key* is 7, the parameter *bufptr* points to the structure *meter_roam*. Specify 50 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_roam,
       2  version FIXED BIN(15),
       2  unused_field FIXED BIN(15),
       2  read_write,
          3  reads FIXED BIN(31),
          3  writes FIXED BIN(31),
          3  retrieve_trans FIXED BIN(31),
          3  update_trans FIXED BIN(31),
          3  non_trans FIXED BIN(31),
          3  windowed FIXED BIN(31),
          3  found_used FIXED BIN(31),
          3  found_free FIXED BIN(31),
          3  disk_reads FIXED BIN(31),
          3  copies FIXED BIN(31),
          3  bef_image_addrs FIXED BIN(31),
       2  release,
          3  calls FIXED BIN(31),
          3  writes FIXED BIN(31),
       2  allocate,
          3  calls FIXED BIN(31),
          3  success FIXED BIN(31),
          3  dynamic FIXED BIN(31),
       2  free_calls,
          3  calls FIXED BIN(31),
          3  others FIXED BIN(31),
       2  before_image,
          3  calls FIXED BIN(31),
          3  converts FIXED BIN(31),
       2  purge_files FIXED BIN(31),
       2  check_calls FIXED BIN(31),
       2  transition,
          3  trans_in FIXED BIN(31),
          3  trans_out FIXED BIN(31);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the version input parameter (see above).

unused_field

OUTPUT. Reserved.

read_write

OUTPUT. A group of fields containing read/write statistics since ROAM initialization.

read_write.reads
OUTPUT. Number of read requests.

read_write.writes
OUTPUT. Number of write requests.

read_write.retrieve_trans
OUTPUT. Number of retrieval transactional accesses.

read_write.update_trans
OUTPUT. Number of update transactional accesses.

read_write.non_trans
OUTPUT. Number of nontransactional accesses.

read_write.windowed
OUTPUT. Number of people currently looking at buffer.

read_write.found_used
OUTPUT. Number of times buffer found in use.

read_write.found_free
OUTPUT. Number of times buffer found on free chain.

read_write.disk_reads
OUTPUT. Number of disk reads required to get buffer.

read_write.copies
OUTPUT. Number of times buffer was copied from another buffer.

read_write.bef_image_addrs
OUTPUT. Number of times a record's before image was read.

release
OUTPUT. A group of fields containing release statistics since ROAM initialization.

release.calls
OUTPUT. Number of release calls.

release.writes
OUTPUT. Number of release calls requiring write operations.

allocate

OUTPUT. A group of fields containing allocation statistics since ROAM initialization.

allocate.calls

OUTPUT. Number of allocate calls.

allocate.success

OUTPUT. Number of times data was in cache.

allocate.dynamic

OUTPUT. Number of dynamic allocations.

free_calls

OUTPUT. A group of fields containing free call statistics since ROAM initialization.

free_calls.calls

OUTPUT. Number of free calls.

free_calls.others

OUTPUT. Number of free calls for other users.

before_image

OUTPUT. A group of fields containing before-image statistics since ROAM initialization.

before_image.calls

OUTPUT. Number of before-image calls.

before_image.converts

OUTPUT. Number of times a page was converted to the before-image state.

purge_files

OUTPUT. Number of file purges.

check_calls

OUTPUT. Number of check calls.

transition

OUTPUT. A group of fields containing buffer transition information since ROAM initialization.

transition.trans_in

OUTPUT. Number of incoming read operations.

transition.trans_out

OUTPUT. Number of outgoing write operations.

Scheduler Information Meters: When *meter_key* is 8, the parameter *bufptr* points to the structure *meter_sch*. *meter_sch* can be used only with Rev. 23.0 and subsequent PRIMOS revisions. Specify 103 for *bufsize* and declare the following structure in your program:

```
DCL 1  meter_sch,
       2  version  FIXED BIN(15),
       2  mode  FIXED BIN(15),
       2  reserved  FIXED BIN(15),
       2  queue_ratios  (1:3) FIXED BIN(15),
       2  level_ratios  (1:5) FIXED BIN(15),
       2  quota_calc_cnt  FIXED BIN(31),
       2  high_queue_ntfy_cnt  (1:5) FIXED BIN(31),
       2  elig_queue_ntfy_cnt  (1:5) FIXED BIN(31),
       2  low_queue_ntfy_cnt  (1:5) FIXED BIN(31),
       2  high_queue_arrv_cnt  (1:5) FIXED BIN(31),
       2  elig_queue_arrv_cnt  (1:5) FIXED BIN(31),
       2  low_queue_arrv_cnt  (1:5) FIXED BIN(31),
       2  high_queue_jobs_cnt  (1:5) FIXED BIN(31),
       2  elig_queue_jobs_cnt  (1:5) FIXED BIN(31),
       2  low_queue_jobs_cnt  (1:5) FIXED BIN(31);
```

version

OUTPUT. Version number of the structure. Returns the value specified for the *version* input parameter (see above).

mode

OUTPUT. Mode of the scheduler.

Returns 0 if the Set_Scheduler _Attributes (SSA) command has either not been invoked since coldstart or was last invoked with no options.

Returns 2 if the Set_Scheduler _Attributes (SSA) command was last invoked with the –SJOB, –QRAT, –PBIAS, or –PRAT options.

The scheduler mode can only be set by the System Administrator, or at the system console. The Set_Scheduler_Attributes command is described in the *Operator's Guide to System Commands*.

reserved

OUTPUT. Reserved for internal use.

queue_ratios

OUTPUT. Returns the CPU ratios for the high-priority, eligibility, and low-priority queues (in that order). The default is –1, –1, 1. A value of –1 represents infinite service; that is, all queued items are taken from that queue before checking the next queue.

level_ratios

OUTPUT. Returns the CPU ratios for the priority levels. Lists from highest priority (level 4) to lowest priority (level 0). The default is 16, 8, 4, 2, 1.

quota_calc_cnt

OUTPUT. Number of times that the quota has been calculated since coldstart. Periodically, PRIMOS automatically calculates the quota for queues that do not have infinite service by using the values shown in *queue_ratios* and *level_ratios*.

high_queue_ntfy_cnt

OUTPUT. Number of processes scheduled off of the high-priority queue since coldstart. When *queue_ratios* indicates non-infinite service for the high-priority queue, this count is subdivided into each of its priority levels. When *queue_ratios* indicates infinite service, each process scheduled off of this queue increments the counter for the highest priority level, regardless of the actual priority level of the process. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to call G$METR twice, then compare the two values for this field.

elig_queue_ntfy_cnt

OUTPUT. Number of processes scheduled off of the eligibility queue since coldstart. When *queue_ratios* indicates non-infinite service for the eligibility queue, this count is subdivided into each of its priority levels. When *queue_ratios* indicates infinite service, each process scheduled off of this queue increments the counter for the highest priority level, regardless of the actual priority level of the process. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to run G$METR twice, then compare the two values for this field.

low_queue_ntfy_cnt

OUTPUT. Number of processes scheduled off of the low-priority queue since coldstart. This notice count is subdivided into each of its priority levels.

high_queue_arrv_cnt

OUTPUT. Total number of processes arrived at the high-priority queue since coldstart. When *queue_ratios* indicates non-infinite service for the

high-priority queue, this count is subdivided into each of its priority levels. When *queue_ratios* indicates infinite service, the highest priority level contains the total count. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to call G$METR twice, then compare the two values for this field.

elig_queue_arrv_cnt

OUTPUT. Total number of processes arrived at the eligibility queue since coldstart. When *queue_ratios* indicates non-infinite service for the eligibility queue, this count is subdivided into each of its priority levels. When *queue_ratios* indicates infinite service, the highest priority level contains the total count. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to call G$METR twice, then compare the two values for this field.

low_queue_arrv_cnt

OUTPUT. Total number of processes arrived at the low-priority queue since coldstart. This count is subdivided into each of its priority levels.

high_queue_jobs_cnt

OUTPUT. Total number (since coldstart) of processes observed waiting on the high-priority queue. Each time PRIMOS calculates the queue quotas, it adds the number of processes observed on this queue to the total of previous observations since coldstart. When *queue_ratios* indicates infinite service, PRIMOS does not observe processes waiting on this queue. When *queue_ratios* indicates non-infinite service, this count is subdivided into each of its priority levels. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to call G$METR twice, then compare the two values for this field.

elig_queue_jobs_cnt

OUTPUT. Total number (since coldstart) of processes observed waiting on the eligibility queue. Each time PRIMOS calculates the queue quotas, it adds the number of processes observed on this queue to the total of previous observations since coldstart. When *queue_ratios* indicates infinite service, PRIMOS does not observe processes waiting on this queue. When *queue_ratios* indicates non-infinite service, this count is subdivided into each of its priority levels. If the queue ratio has been changed since coldstart, the total number shown in this field may not be very useful; what is often more useful is to call G$METR twice, then compare the two values for this field.

low_queue_jobs_cnt

OUTPUT. Total number (since coldstart) of processes observed waiting on the low-priority queue. Each time PRIMOS calculates the queue quotas, it adds the number of processes observed on this queue to the total of previous

observations since coldstart. This count is subdivided into each of its priority levels.

Discussion

G$METR can return metering information about the system as a whole, the file system, interrupt processes, memory, disks, ROAM (Recovery-Oriented Access Method), the scheduler, or individual users. Each category of information requires a separate call to G$METR. The size and structure of the caller's return buffer depends on the *meter_key* chosen. Process aborts are inhibited during the fetch to ensure consistency of the information returned.

For some G$METR meters (for example, *meter_sch*) the recommended practice is to run G$METR twice during a period of known duration and unchanged scheduler parameters, then compare the two resulting values for each meter field.

Loading and Linking Information

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

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

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

User Terminal I/O

3

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 *PRIMOS 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 *PRIMOS 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

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. *Subroutines Reference IV: Libraries and I/O* 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	Read a character.
C1IN$	Read a character.
C1NE$	Read a character, suppressing echo.
CL$GET	Read a line.
CNIN$	Read a specified number of characters.
COMANL	Read a line into a PRIMOS buffer.
ECL$CC	Supervise editing of terminal or command file input; callable from C.
ECL$CL	Provide interface to ECL$CL from non-C programs.
RDTK$$	Parse a command line.
T1IB	Read a character (function).
T1IN	Read a character (procedure).
TIDEC	Read a decimal number.
TIHEX	Read a hexadecimal number.
TIOCT	Read an octal number.

C1IN

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$

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$

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

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 data type 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 object of this program is to read three        }
{strings from the terminal and display them in      }
{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$

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 *PRIMOS 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

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.

ECL$CC

This routine supervises the editing of input from a terminal or a command file, for programs written in C. (Use ECL$CL for calling from other languages.)

Usage

(Declaration in the C language)

```
#define vstr(size)  struct { short len;
                     char data[size+1]; }
typedef vstr(160)   vstr_comline;
typedef vstr(50)    vstr_prompt;

extern unsigned short int       /* [R] comi_switch */
ECL$CC  ( vstr_comline *,    /* [i/o] comline */
        int,                 /* [I] comline_size */
        vstr_prompt *,       /* [i] prompt     */
        int,                 /* [I] show_prompt */
        short * );           /* [o] code */
```

comi_switch = ECL$CC(&*comline*, *comline_size*,
&*prompt*, *show_prompt*, &*code*);

Parameters

comline

INPUT –> OUTPUT. A buffer used for input and output. On input, it contains a string to be inserted into the command line for the user to edit. Because most applications use ECL to ask for a new command, this is normally a null string. The cursor is placed at the beginning of the inserted string. On output, the finished command line is returned to *comline*.

comline_size

INPUT. The maximum number of characters that can be stored in comline.

prompt

INPUT. The text of the prompt. It can contain only printable characters and the placeholder character, #. To make the # character appear in the text of the prompt, specify ## where the character is to appear. If *show_prompt* is set to 1, the placeholder character is replaced by the current command history entry.

show_prompt

INPUT. Specify a value of 1 if ECL is to display the prompt. Specify a value of 0 if the calling program has already displayed the prompt.

code

OUTPUT. The status code. Possible values are

0	The call to ECL$CC was completed without error.
–1	Input was aborted by the ECL command abort_cmd.

comi_switch

OPTIONAL RETURNED VALUE. Zero if input was read from the user terminal, and nonzero if input was read from a command input file. PRIMOS uses this value to suppress abbreviation expansion of a command line read from a command input file.

Discussion

ECL$CC supports EDIT_CMD_LINE (ECL), a terminal interface and command line editor. ECL$CC enables a user program written in C to read a single command line from a terminal or a command file and allows extensive editing of the command line.

ECL$CC can insert a string into the command line before the user begins to edit. Applications that use ECL to ask for a new command normally insert a null string for the prompt. Applications that want the user to type something might insert a string that provides information about what the user is to type. For example, an application that requests the user to type a number might issue the prompt

```
Which user:
```

and insert the string "(enter the user's number)" into the command line following the prompt. The resulting display would be

```
Which user: (enter the user's number)
```

The cursor appears on the first character of the inserted string, '('. When the user types a number, the inserted string disappears, unless the user chooses to edit the inserted string.

Most user programs do not need to know whether the command comes from a file or from the terminal. For this reason, most user programs can declare ECL$CC with the void type specifier and call ECL$CC as a procedure rather than as a function.

ECL$CC is designed to be called from C. Programs in other languages can call ECL$CL, which in turn calls ECL$CC.

If your program uses ECLCC, ECLLIB.RUN must be in your entrypoint search rules.

Example

The following code shows how ECL$CC can display a prompt. The code does not insert a string into the command line.

```
unsigned short  comi_switch;
vstr_comline  comline;
int           comline_size;
vstr_prompt   prompt;
int           show_prompt;   /* 0 or 1 */
short         code;
strcpy(comline.data, "");
comline.len = strlen(comline.data);
                  /* Initialize comline        */
                  /* to null PL/1-style string */

comline_size = 160;
strcpy(prompt.data, "MY_PROMPT: ");
prompt.len = strlen(prompt.data);
show_prompt = 1;
comi_switch = ECL$CC(&comline, comline_size,
                     &prompt, show_prompt, &code);
```

Loading and Linking Information

V-mode and I-mode: With shared or unshared libraries, load ECL$LIB.BIN. With unshared libraries, load ECL$LIB.BIN.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

ECL$CL

This routine supervises the editing of input from a terminal or command file, for programs not written in C.

Usage

```
DCL ECL$CL ENTRY (CHAR (*)VAR, FIXED BIN(15),
                  FIXED BIN(15) [, CHAR (*)VAR,
                  BIT(1) ALIGNED] );

CALL ECL$CL (comline, comline_size, code [, prompt, show_prompt] );
```

Programs that need to know whether input is from a user terminal or a command input file can call ECL$CL as function, using statements of the following form.

```
DCL ECL$CL ENTRY (CHAR (*)VAR, FIXED BIN(15),
                  FIXED BIN(15) [, CHAR (*)VAR,
                  BIT(1) ALIGNED] ) RETURNS (FIXED BIN(15));

comi_switch = ECL$CL (comline, comline_size, code
                  [, prompt, show_prompt] );
```

Parameters

comline

INPUT -> OUTPUT. A buffer used for input and output.

On input, the contents of the buffer depend on whether ECL$CL is called with or without the *prompt* and *show_prompt* arguments.

If ECL$CL is called with the *prompt* and *show_prompt* arguments, *comline* contains a string to be inserted into the command line for editing. Normally, this is a null string. The cursor is placed at the beginning of the string.

If ECL$CL is called without the *prompt* and *show_prompt* arguments, *comline* must contain the prompt already displayed by the application prior to the call to ECL$CL; ECL's display manager uses the prompt to maintain the screen properly.

On output, the finished command line is returned to *comline*.

comline_size

INPUT. The maximum number of characters that can be stored in *comline*.

code

OUTPUT. The status code. Possible values are

0	The call to ECL$CL was completed without error.
–1	Input was aborted by the ECL command abort_cmd.

prompt

OPTIONAL INPUT. The text of the prompt. It can contain only printable characters and the placeholder character, #. To make the # character appear in the text of the prompt, specify ## where the character is to appear. If *show_prompt* is true, the placeholder character is replaced by the current command history entry. If ECL$CL is called without the *prompt* and *show_prompt* arguments, the calling program must pass the prompt using the *comline* parameter.

show_prompt

OPTIONAL INPUT. Specify a value of '1'b if ECL is to display the prompt. Specify a value of '0'b if the calling program has already displayed the prompt. The call must include this parameter if it includes the *prompt* parameter.

comi_switch

OPTIONAL RETURNED VALUE. Zero if input was read from the user terminal, and nonzero if input was read from a command input file. PRIMOS uses this value to suppress abbreviation expansion of a command line read from a command input file. However, most user programs do not need to know whether the command came from a file or from the terminal. For this reason, most user programs can omit the RETURNS(FIXED BIN(15)) portion of the declaration and call ECL$CL as a procedure rather than as a function.

Discussion

ECL$CL provides an interface to ECL$CC from PL/I and other high-level languages, except C. ECL$CC supervises the editing of command line input from a terminal or command file. ECL$CC should be called only from C.

If your program uses ECLCL, ECLLIB.RUN must be in your entry point search rules.

Examples

1. The calling program displays the prompt.

```
prompt = 'Enter result ';
call tnoua(prompt, length(prompt));
comline = prompt;
call ecl$cl(comline, comline_size, code);
```

or

```
prompt = 'Enter result ';
call tnoua(prompt, length(prompt));
comline = '';
call ecl$cl(comline, comline_size, code, prompt, '0'b);
```

2. ECL$CL displays the prompt. The calling program does not insert a string.

```
prompt = 'Enter result ';
comline = '';
call ecl$cl(comline, comline_size, code, prompt, '1'b);
```

3. Sample program to edit or create a global variable.

```
edit_gvar:  procedure (com_args, code);
 /* EPF calling conventions */

    dcl  com_args        char(256) var;
    dcl  code            fixed bin(15);

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

    %replace MAXVARSIZE by 1024;

    dcl  er$print entry(fixed bin(15), char(*) var,
                        fixed bin(15), char(*) var,
                        char(*) var);
    dcl  gv$get   entry(char(*) var, char(*) var,
                        fixed bin(15), fixed bin(15));
    dcl  gv$set   entry(char(*) var, char(*) var,
                        fixed bin(15));
    dcl  ecl$cl   entry(char(*) var, fixed bin(15),
                        fixed bin(15), char(*) var,
                        bit(1) aligned);

    dcl  gvar_name char(256) var;
    dcl  prompt char(256) var;
```

```
dcl  var_value char(MAXVARSIZE) var;
dcl  LINKAGE_FAULT$ condition;
dcl  translate builtin;

/*
 * Get current value of indicated variable.  If not
 * defined, then initialize to null string.  Global
 * variable storage must be defined.  The GVAR name
 * must be uppercase.
 */
gvar_name = translate(com_args,
                      'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                      'abcdefghijklmnopqrstuvwxyz');
call gv$get(gvar_name, var_value, MAXVARSIZE, code);
select (code);
     when (0)           /* Okay */
          ;
    when (e$unop)       /* No gvar storage defined */
        call er$print(k$nrtn, ssc$errd, e$null,
               'No global variable storage defined.',
                 'EDIT_GVAR');
    when (e$fntf)               /* No such variable */
        var_value = '';         /* Create a new one */
    otherwise
        call er$print(k$nrtn, ssc$errd, code,
                 'Could not continue.', 'EDIT_GVAR');
end;

/*
 * Pass value to ECL$CL for editing.  If user hits
 * ^G (aborts), the value of the variable does not
 * change.  Trap problems if
 * ECL$LIB interface is not installed.
 */
on condition(LINKAGE_FAULT$)
    call er$print(k$nrtn, ssc$errd, e$null,
              'Edit_Cmd_Line package not installed.',
               'EDIT_GVAR');

/*
 * var_value has been set above to the current value
 * of the variable, or the null string if it is a new
 * variable.  This will be inserted.  Prompt will be
 * name of global variable to be modified.
 */
prompt = gvar_name || ': ';
call ecl$cl(var_value, MAXVARSIZE, code, prompt,
            '1'b);

/* User had typed ^G to abort edit operation */
if (code ^= 0) then stop;
```

```
    /*  Set new value of global variable. */
    call gv$set(gvar_name, var_value, code);
    call er$print(k$irtn, ssc$errd, code,
                  'GVAR not set.', 'EDIT_GVAR');
    stop;
end;
```

Loading and Linking Information

V-mode and I-mode, shared or unshared libraries: Load ECL$LIB.BIN.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

RDTK$$

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 zero 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

E$OK	No errors.
E$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 character.

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
AOOO1
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

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 can be called from PMA to load a character from the terminal into Register A. It 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

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

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

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

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*
ER$PRINT	Print a standard error message from PRIMOS or a PRIMOS subsystem.
IOA$	Provide free-format output.
IOA$ER	Provide free-format output, for error messages.
TNOU	Write characters to terminal, followed by newline character.
TNOUA	Write characters to terminal.
TODEC	Write a signed decimal number.
TOHEX	Write a hexadecimal number.
TONL	Write a newline character.
TOOCT	Write an octal number.
TOVFD$	Write a decimal number, without spaces.
T1OB	Write one character from Register A.
T1OU	Write one character.

ER$PRINT
ER$PNT

This routine prints error messages from PRIMOS and specified PRIMOS subsystems on the user terminal.

Usage

```
DCL ER$PRINT ENTRY (FIXED BIN(15), CHAR(*)VAR,
                    FIXED BIN(15),
                    CHAR(*)VAR, CHAR(*)VAR);

CALL ER$PRINT (key, sscode, ercode, text, name);
```

Parameters

key

INPUT. The action that ER$PRINT is to take after printing a 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.

sscode

INPUT. The subsystem for which ER$PRINT is to print an error message. Specify a code that designates a PRIMOS subsystem. Subsystem codes are defined in the file ERRORMSGHDLR.INS.*language*, where the *language* suffix denotes the programming language. Some of the possible subsystem codes are

SSC$ERRD	ERRD messages (ERRD$)
SSC$SYNC	Event synchronizers (SYNC$)
SSC$TIMER	Timers (TIMER$)
SSC$ISC	InterServer Communications (ISC$)

ercode

INPUT. The return code from the subroutine that produced the error. If *ercode* is E$OK or invalid, ER$PRINT prints nothing and control returns immediately to the calling program.

text

INPUT. A message to be printed following the subsystem error message. The maximum number of characters is 256.

name

INPUT. The name of the calling routine. The maximum number of characters is 256.

Discussion

ER$PRINT finds an error message in a message file in the SYSOVL directory or in the PRIMOS internal message table, and displays the message on the terminal. ER$PRINT can also display qualifying text for the error message and the name of the routine that calls ER$PRINT, if the text and the name are specified in the call to ER$PRINT. ER$PRINT is similar in function to ERRPR$, except that ER$PRINT can return messages for a particular PRIMOS subsystem that you specify.

To find an error message, ER$PRINT first looks in the SYSOVL directory for a message file, as specified by the subsystem name appended with _ERROR_TABLE. For example, the synchronizer message file is specified by SYNC$_ERROR_TABLE. Note that by convention, the names of PRIMOS subsystems end with a dollar sign ($). If the specified SYSOVL file exists, ER$PRINT displays the message in the SYSOVL message file that corresponds to the error code in *ercode*.

If the SYSOVL file does not exist, ER$PRINT looks for the message in the PRIMOS internal message table. The messages in the PRIMOS internal message table are in English. If ER$PRINT cannot find the message in the PRIMOS internal message table, it prints the values of *ercode* and *sscode*.

Programs that call ER$PRINT must insert the file SYSCOM>ERRORMSGHDLR.INS.*language*, where *language* is a suffix specifying the program's language. Programs should use the values defined in the insert file rather than the numeric values or strings to which the values correspond.

Loading and Linking Information

The dynamic link for ER$PRINT is in PRIMOS.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

IOA$

IOA$ provides free-format terminal output.

Usage

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

The form of the declaration for IOA$ depends on the number and types of arguments specified in the call to IOA$.

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.

Discussion

IOA$ provides free-format output to the terminal. Most application programs can use the standard output package provided with the programming language. For example, C programmers should use the standard C procedure *printf*, on which IOA$ is based. Pascal programmers should use the standard Pascal procedure *write*.

However, systems programs can benefit from the efficiency and flexibility of IOA$. The format of the IOA$ template is simple and can be constructed even 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.

A simple example follows below. This statement 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:prec.scaleZRtype
```

The notations *fw*, *prec*, *.scale*, and *type* each stand for a single character or possibly (in the case of *fw* and *.scale*) 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 consists of *fw* 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. For nonfloating numbers, 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

For floating point numbers, the possible values are

1	Signed 32-bit integer
2	Signed 64-bit or 128-bit integer

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

.scale Specifies the number of digits to display to the right of the decimal point. Note the required period. The .scale specifier cannot follow immediately after the percent symbol specifier.

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.
E	Output the next argument as a decimal number in exponential form. The maximum number of digits that IOA$ displays to the right of the decimal point is 8. When you specify .8 for .SCALE, IOA$ displays the number as d.ddddddddE dd, where the final "dd" is a positive exponent. A negative exponent is expressed as a final "–dd". If the exponent contains more than two digits, the final "dd" is extended to the right. Examples of numbers in exponential format are: 1.01386568E168 (three-digit positive exponent) and 7.66629847E–66 (two-digit negative exponent).
F	Output the next argument as a decimal number in fixed-point notation. The maximum number of digits that IOA$ displays in this format is 10. If a number is greater than 2147483647.0 after scaling, IOA$ displays the number in E format. For example, if you specify a control code of %:1.5F to display the number 3.0E5, IOA$ displays the number as 3.00000E05 rather than as 300000.00000.
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 and should be halfword aligned. 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 value:  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

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

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 0 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

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

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

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

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

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

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$

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

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

T1OU writes a character to the user terminal.

Usage

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

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 a newline character, a return and a 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$	Inhibit or enable BREAK function.
CO$GET	Return information about command output settings.
COMI$$	Switch input between the terminal and a file.
COMO$$	Switch output between the terminal and a file.
DUPLX$	Control the way PRIMOS treats the user terminal.
ERKL$$	Read or set the erase and kill characters.
QUIT$	Determine if there are pending quits.
TTY$IN	Check for unread terminal input characters.
TTY$OUT	Check whether there are characters in user's terminal output buffer for a calling process.
TTY$RS	Clear the terminal input and output buffers.

BREAK$

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 decrements 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

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$$

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$$

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. The file must be in the current directory. Do not specify a full pathname.

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, ER$PRINT, or IOA$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$

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. This bit is meaningful only with half duplex (bit 1 set).
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.)
6	:002000	Handle reverse channel functionality. (See AMLC Functions, below.)

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

Bit	*Mask*	*Meaning if Bit is Set*
7	:001000	Check for certain error conditions: • Overflow of the input buffer • Parity error If one of these conditions is present, the character found is replaced with 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). These bits have no meaning on systems configured with more than 255 terminal users.

Note DUPLX$ returns 0 as the internal buffer number if the number is greater than or equal to 256. For this reason, always use AS$LIN or AS$LST to retrieve the buffer number, parity error, and echo setting when the internal buffer number of a line is greater than or equal to 256. AS$LIN and AS$LST can also be used to retrieve this information when the line's buffer number is less than 256.

Discussion

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.

When the terminal is configured for full duplex, bit 2 of the terminal configuration word is ignored. When the terminal is configured for half duplex, the line protocol indicates whether to echo LF after CR.

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.)
```

DUPLX$ 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.

ERKL$$

This routine 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

E$OK	No errors.
E$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. I/O facilities of all language processors 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$

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

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$OUT

Check whether there are any characters in the user's terminal output buffer for a calling process.

Usage

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

tty_has_output = TTY$OUT ( );
```

Parameters

tty_has_output
RETURNED VALUED. True ('1'b) if there are any characters in the terminal output buffer for the calling process. False ('0'b) otherwise.

Discussion

TTY$OUT checks whether there are any characters in the terminal output buffer for the calling process. TTY$OUT does not affect the state or contents of the terminal output buffer. The user can call TTY$OUT at any time.

When TTY$OUT is called by a noninteractive process, TTY$OUT always returns '0'b, whether or not a command input file is active.

If the user presses CONTROL–P while TTY$OUT is executing, TTY$OUT returns TRUE ('1'b), indicating that there are characters in the terminal output buffer. In this case, the return value is incorrect, because CONTROL–P flushes the terminal output buffer and causes a quit to PRIMOS.

It is not possible to call TTY$OUT directly from a program written in FTN, because FTN programs cannot call functions that do not have 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.

Effective for PRIMOS Revision 22.0 and subsequent revisions.

TTY$RS

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 codes.

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.

Memory Allocation

4

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 III: Command Environment* 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 a pointer argument. 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$S	Allocate memory on the current stack.
MM$MLPA	Make the last page of a segment available.
MM$MLPU	Make the last page of a segment unavailable.
STR$AL	Allocate user-class dynamic memory.
STR$AP	Allocate process-class dynamic memory.
STR$AS	Allocate subsystem-class dynamic memory.
STR$AU	Allocate user-class dynamic memory.
STR$FP	Free process-class dynamic memory.
STR$FR	Free user-class dynamic memory.
STR$FS	Free subsystem-class dynamic memory.
STR$FU	Free user-class dynamic memory.

ALOC$S

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 deallocated. 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.

MM$MLPA

This routine makes the last page of a segment available.

Usage

DCL MM$MLPA ENTRY (FIXED BIN, FIXED BIN);

CALL MM$MLPA (*segment, code*);

Parameters

segment

INPUT. Segment containing the page to be made available. Must be in the range from 2048 through 4095 (octal 4000 through 7777).

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BSGN	Segment out of range (not between 2048 and 4095).
E$NOSG	Segment not in use.
E$BDAT	Page not currently out of bounds.

Discussion

MM$MLPA enables use of the last page of a segment that MM$MLPU made unavailable. Refer to the MM$MLPU description for more information about that 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: No special action.

MM$MLPU

This routine makes the last page of a segment unavailable.

Usage

DCL MM$MLPU ENTRY (FIXED BIN, FIXED BIN);

CALL MM$MLPU (*segment, code*);

Parameters

segment

INPUT. Segment containing the page to be made unavailable. Must be in the range from 2048 through 4095 (octal 4000 through 7777).

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BSGN	Segment out of range (not between 2048 and 4095).
E$NOSG	Segment not in use.
E$BDAT	Page already in use.
E$DKFL	Paging disk is full.

Discussion

When MM$MLPU makes a page unavailable, subsequent attempts to access the page result in the OUT_OF_BOUNDS$ condition. This could be useful for certain memory allocation schemes.

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.

STR$AL

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$OK	No error
E$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

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 reinitialize 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

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 subsystems supplied by Prime *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$OK	No error
E$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

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

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 reinitialize 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

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$OK	No error
E$FRER	Invalid free request
E$HPER	Corrupted heap

Discussion

For performance reasons, PRIMOS deallocates fully allocated segments immediately and deallocates partially allocated segments at program termination.

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

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$OK	No error
E$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

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$RA	Allocate space for EPF function return information.
ALS$RA	Allocate space and set value of EPF function return information.
FRE$RA	Deallocate space for EPF function return information.

ALC$RA

This routine allocates space for EPF (Executable Program Format) function return information.

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 III: Command Environment* 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 FIXED BIN(15),    /* Rev. 21.0 */
                      2, 3 FIXED BIN(31),
                         3 FIXED BIN(31),
                         3 FIXED BIN(31),
                         3 FIXED BIN(31),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(11),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(14),
                         3 FIXED BIN(15),
                         3 FIXED BIN(15),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(1),
                         3 BIT(13),
                         3 FIXED BIN(31), /* Rev. 21.0 */
                         3 FIXED BIN(31), /* Rev. 21.0 */
                         3 FIXED BIN(31), /* Rev. 21.0 */
                         3 FIXED BIN(31), /* Rev. 21.0 */
                   1, 2 BIT(1),
                      2 BIT(15),
                   PTR);

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

This interface is appropriate for Rev 21.0 and subsequent revisions. Users of PRIMOS revisions prior to 21.0 should set the version argument to 1 and omit the fields indicated. The 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; 2 at Rev. 21.0 and subsequent revisions).
vcb_ptr	Pointer to CPL local variables.
vcb_ptr_ext	Extension to the vcb_ptr. This field is used only for Rev. 21.0 and subsequent revisions.
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.
created_after_date	If nonzero, then the command processor has found something created after the given date. This field is used only for Rev. 21.0 and subsequent revisions.
created_before_date	If nonzero, then the command processor has found something created before the given date. This field is used only for Rev. 21.0 and subsequent revisions.
accessed_after_date	If nonzero, then the command processor has found something accessed after the given date. This field is used only for Rev. 21.0 and subsequent revisions.
accessed_before_date	If nonzero, then the command processor has found something accessed before the given date. This field is used only for Rev. 21.0 and subsequent revisions.

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

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;
```

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$RA, your program must set the version number of *rtn_fcn_struc* to 0 and copy the value of that structure into *value_str*. Then the interface sets *rtn_fcn_ptr* in its main entrypoint's calling sequence and returns to the calling program. Refer to the *Advanced Programmer's Guide III: Command Environment* for a further discussion of ALC$RA and 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.

ALS$RA

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

This routine deallocates 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$SGS	Return maximum number of dynamic segments.
ST$SGS	Return maximum number of static segments.
TL$SGS	Return highest segment number.

DY$SGS

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

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

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 through 5777 octal (from 2048 through 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.

Program Control

5

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 *PRIMOS 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. ER$PRINT 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 *Subroutines Reference II: File System* and in the *Advanced Programmer's Guide I: BIND and EPFs.*

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 *PRIMOS 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 be sent only 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$E	Call a new command level after an error.
COMLV$	Call a new command level.
EXIT	Return to PRIMOS.
ICE$	Initialize the command environment.
KLM$IF	Return serialization data.
SETRC$	Record command error status.
SS$ERR	Signal an error in a subsystem.

CMLV$E

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$

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

This routine 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$

This routine initializes the command environment.

Usage

DCL ICE$ ENTRY (CHAR(80) VAR, FIXED BIN(15));

CALL ICE$ (*args, code*);

Parameters

args

Command arguments. You may specify –SERVER, or a null argument. Any argument other than –SERVER is treated as a null argument, and no error is returned.

code

The standard error code. Error codes are never returned to the user, because the call to ICE$ terminates the calling program.

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

Discussion

ICE$ resets your environment to its initial state. When specified with no arguments, ICE$ closes all open files, including the command output file and the current program file, resets search rule lists to the system defaults, and deallocates all user resources, such as private dynamic and static segments, virtual circuits, buffers, and slave processes.

Beginning at Rev. 22.0, ICE$ also releases all semaphores and RJE devices, resets Information echo delay, releases all assigned devices (except when called by User 1), resets the erase & kill characters to the default, enables terminal output, and enables messages. ICE$ does not reset server names.

When specified with the –SERVER argument, ICE$ performs all of the operations listed above, and in addition closes all ISC sessions, reinitializes the server's SessionRequestPending synchronizer, logs out all child processes of the caller, and deletes all timers and synchronizers. When ICE$ with the –SERVER argument is called by a child process, the –SERVER argument is ignored.

The program that calls ICE$ is terminated. If you are working in a subdirectory when ICE$ is executed, you are returned to your origin directory.

Programs using pre-Rev. 22.0 versions of ICE$ do not need to be modified for use with Rev. 22.0.

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.

KLM$IF

This routine enables a program to obtain serialization data from a specified file.

Usage

```
DCL KLM$IF ENTRY (CHAR(*)VAR, POINTER, POINTER,
                  POINTER, POINTER, POINTER,
                  FIXED BIN);

CALL KLM$IF (pathname, std_info_ptr, cmp_info_ptr, dst_info_ptr,
             ins_info_ptr, doc_info_ptr, code);
```

Parameters

pathname

INPUT. Specifies the name of the file for which serialization information will be returned. The Discussion section contains more information about this parameter.

std_info_ptr

INPUT –> OUTPUT. Pointer to the structure that will contain standard information. The data area for the information returned must be at least 75 halfwords long.

cmp_info_ptr

INPUT –> OUTPUT. Reserved; must be a null pointer.

dst_info_ptr

INPUT –> OUTPUT. Pointer to the structure that will contain distribution information. A null pointer can be specified if distribution information is not required. The data area for the information returned must be at least 28 halfwords long.

ins_info_ptr

INPUT –> OUTPUT. Reserved; must be a null pointer.

doc_info_ptr

INPUT –> OUTPUT. Reserved; must be a null pointer.

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BNAM	Illegal pathname specified.
E$NTFD	Pathname identifies an illegal file type.
E$NDAM	EPF specified is not a DAM file.
E$FNTS	Segment 0 file not found in segment directory.
E$BVER	Unsupported structure version number.
E$BPAR	Null pointer specified for the parameter *std_info_ptr*.

Structure Description

The parameter *std_info_ptr* points to the structure *ki_standard_info*, shown below.

```
DCL 1  ki_standard_info,
       2  ki_version FIXED BIN,
       2  ki_product_name CHAR(20) VAR,
       2  ki_revision CHAR(20) VAR,
       2  ki_serial_number CHAR(20) VAR,
       2  ki_licensee CHAR(40) VAR,
       2  ki_expiry_date FIXED BIN(31),
       2  ki_primos_base_rev CHAR(10) VAR,
       2  ki_library_base_rev CHAR(10) VAR,
       2  ki_ucode_base_rev CHAR(10) VAR;
```

ki_version

INPUT. Version number of this structure. Must be set to 1.

ki_product_name

OUTPUT. Name of the product.

ki_revision

OUTPUT. Revision of the product.

ki_serial_number

OUTPUT. Serial number of the product.

ki_licensee

OUTPUT. Name of licensed user of product.

ki_expiry_date

OUTPUT. Software expiration date, in file-system date format. File-system date format is described in Appendix C.

ki_primos_base_rev

OUTPUT. The earliest revision of PRIMOS that supports this product.

ki_library_base_rev

OUTPUT. The earliest library revision that supports this product.

ki_ucode_base_rev

OUTPUT. The earliest microcode revision that supports this product.

The parameter *cmp_info_ptr* is a null pointer to a structure that is currently reserved.

The parameter *dst_info_ptr* points to the structure *ki_distribution_info*, shown below.

```
DCL 1  ki_distribution_info,
       2  ki_organization CHAR(20) VAR,
       2  ki_individual CHAR(6) VAR,
       2  ki_issue_date FIXED BIN(31),
       2  ki_order_number CHAR(8) VAR,
       2  ki_csm_number CHAR(10) VAR;
```

ki_organization

OUTPUT. Name of the organization distributing the software.

ki_individual

OUTPUT. Name of the individual responsible for software distribution.

ki_issue_date

OUTPUT. Software distribution date, in file-system date format. File–system date format is described in Appendix C.

ki_order_number

OUTPUT. Order number of the distributed software.

ki_csm_number

OUTPUT. Customer service maintenance number for the product license.

The parameter *ins_info_ptr* is a null pointer to a structure that is currently reserved.

The parameter *doc_info_ptr* is a null pointer to a structure that is currently reserved.

Discussion

KLM$IF can use a simple filename, supplied by a program, and system search rules to obtain serialization data from an installed product (in CMDNC0) of that name. By specifying the full or relative pathname, a program can obtain serialization data from any file on the system.

Loading and Linking Information

V-mode and I-mode shared: Not available. Use the unshared version.

V-mode and I-mode with unshared libraries: Link with LIB>KLM$IF.

R-mode: Not available.

Effective for PRIMOS Revision 21.0 and subsequent revisions.

SETRC$

This routine returns to the system a user-specified status code when the calling program exits.

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

SETRC$ records the code that you specify as *severity_code*. Later, when the program exits, the system regards this code as the error status. SETRC$ does not cause an immediate return to the calling software.

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.

When an EPF sets a static mode error code (either by calling a static mode program, or by calling SETRC$) the PRIMOS prompt that appears when the EPF exits reflects the more severe of the two codes — the static mode error code or the EPF's program status code. Thus, a static mode error overrides a program status warning.

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

This routine signals an error in a subsystem. It is intended to terminate the program immediately if it is invoked from a command input file.

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 using the program interactively, 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$$	Restore an R-mode executable image.
RESU$$	Restore and resume an R-mode executable image.
SAVE$$	Save an R-mode executable image.

REST$$

This routine reads R-mode executable code into memory from a file in the current directory.

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.

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).

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 *Subroutines Reference II: File System*, 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.

RESU$$

This routine restores R-mode executable code from a file in the current directory, 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 *Subroutines Reference II: File System*, 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$$

This routine saves an R-mode executable image as a file in the current directory.

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$CN	Switch logout notification on or off.
LON$R	Read logout notification information.
PHNTM$	Start a phantom process.

LON$CN

This routine is used to turn off, or turn on, logout notification.

Usage

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

CALL LON$CN (key);
```

Parameters

key

INPUT. Software interrupt status key:

0	Notify off
1	Notify on

Discussion

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.

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

This routine 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	Message left on queue

code

OUTPUT. Standard error code.

E$OK	No error
E$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$

This routine allows a process to start a phantom using either a command input file or a CPL file.

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; non-CPL programs must not have a .CPL suffix.

name_len

INPUT. The number of characters in *filename*.

funit

INPUT. The file unit on which to open the phantom file. This argument is used only by COMI phantoms. CPL phantoms ignore this argument. Valid file unit numbers range from 1 through 128.

phant_user

OUTPUT. The user number of the phantom.

code

OUTPUT. Standard error code. The possible values are

E$OK	The call to PHNTM$ was completed without error.
E$BUNT	The *funit* value was not within the valid range (1–128).

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.

Discussion

PHNTM$ replaces the obsolete subroutine PHANT$. PHANT$ is described in Appendix D.

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

CH$FX1

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

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

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

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 octal 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$DQS	Convert binary date to quadseconds.
CV$DTB	Convert ASCII date to binary format.
CV$FDA	Convert binary date to ISO format.
CV$FDV	Convert binary date to visual format.
CV$QSD	Convert quadsecond date to binary format.

CV$DQS

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 file–system date format (FS–date). The format of a 32-bit encoded FS-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 bit-encoded FS–date and converts it to absolute quadseconds since January 1, 1901 midnight (01–01–01.00:00:00).

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

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

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 file-system date format (FS-date) returned. The format of a 32-bit encoded FS-date is described in Appendix C.

code

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

E$OK	No error
E$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-date (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-date format, you can perform comparisons on them.

If you need to obtain the current date and time in FS-date 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

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 file-system date format (FS-date). The format of a 32-bit encoded FS-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-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

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 file-system date format (FS-date). The format of a 32-bit encoded FS-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-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

CV$QSD converts a date and time in quadsecond form into file-system date format. One quadsecond equals 4 seconds. CV$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 file-system date format (FS-date). The format of a 32-bit encoded FS-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 bit-encoded FS-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

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:

- The table must fit in a 64K halfword (128K byte) segment.
- All entries must be the same size.
- 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 values:

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$

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), CHAR(16) VAR);

CALL ENCRYPT$ (*encrypted_password, unencrypted_password*);

Parameters

encrypted_password

OUTPUT. The encrypted value of the unencrypted password.

unencrypted_password

INPUT. An ASCII login validation password up to 16 characters long.

Discussion

Login validation passwords may contain any characters other than PRIMOS reserved characters. (See the *PRIMOS 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

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

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

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$

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$

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

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

UID$BT 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 a recent date and time, in file-system date format (FS-date), with a 16-bit counter. (The format of a 32-bit encoded FS-date is described in Appendix C.) Note that the date and time string is used for uniqueness; it may not necessarily be the correct date and time. 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

Given a unique bit string, UID$CH 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.

Condition Mechanism

7

This chapter describes subroutines used in the implementation of the condition mechanism. The first part of this chapter describes subroutines used to signal and catch conditions. The second part describes three subroutines used to control automatic signalling of the EXIT$ condition. The third part describes the data structure formats associated with the condition mechanism. Most programs do not use these data structures.

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.
- Enables 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 condition handler, known as an **on-unit**. PRIMOS finds the on-unit by searching the process's stack for frames that have predefined on-units that can handle that named condition. If PRIMOS finds an on-unit, the on-unit is invoked.

The subroutines described in this chapter enable the programmer to create and use on-units. These features are available to programmers using all languages supplied by Prime. The descriptions below use mostly PL/I terminology, with special advice for FORTRAN users.

Within any procedure, users can create on-units for as many conditions as circumstances require. These conditions can be standard PRIMOS conditions or nonstandard conditions signalled by subroutines described in this chapter.

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.

On-unit Actions

An on-unit can take a variety of actions, including the following:

- Terminate the program by means of a nonlocal GOTO, passing control back to the main program, so that it can call EXIT and return to PRIMOS level.
- Run diagnostic routines and then terminate the program (as above).
- Repair the problem that caused the error condition and have the program resume execution from the point of interruption.
- Ignore the error condition and resume running the program.
- Transfer control to some predetermined spot in the program, possibly in a procedure different from the one that raised the error condition.
- Pass control back to the condition mechanism and tell it to hunt for another on-unit.
- Print messages and then do any of the above.
- Print messages and/or run diagnostic routines and then transfer control back to the user at the terminal (as the system default on-unit does).

Creating and Using On-units

On-units can be procedures or PL/I begin blocks. A begin block results from a PL/I ON statement. Procedures are created by the following subroutines: MKONU$, MKON$F, and MKON$P.

The use of these subroutines is the only way to create an on-unit in a non-PL/I environment.

An on-unit can be invalidated by the PL/I REVERT statement or by using the following subroutines:

RVONU$
RVON$F

The condition mechanism is activated whenever a condition is raised. A condition is raised implicitly by some exception being detected during regular program execution. A condition may be raised explicitly by the PL/I SIGNAL statement or by a call to the following subroutines:

SIGNL$
SGNL$F

The system finds the correct on-unit by searching backwards through the call stack until it encounters an appropriate procedure activation. An appropriate procedure activation is one that has previously created an on-unit for the condition. If an appropriate procedure activation is not found, but if an on-unit for the special condition ANY$ exists, 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 lists the condition mechanism subroutines and summarizes their functions.

Table 7–1. Condition Mechanism Subroutines

Action	*Programming Language (1)*			
	FTN	F77, C, Pascal	PL/I	PMA
Create an on-unit	MKON$F	MKON$P	(4)	MKONU$(2)
Signal a condition	SGNL$F	SGNL$F	SIGNL$	SIGNL$
Cancel (revert) an on-unit	RVON$F	RVON$F	RVONU$(3)	RVONU$
Nonlocal GOTO	PL1$NL	PL1$NL(6)	(4)	PL1$NL
Make PL/I-compatible label	MKLB$F	MKLB$F(5) (6)	(4)	MKLB$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 the Examples of Programs section later in this chapter.
2. 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.
3. Use the language-supplied REVERT statement for PL/I predefined conditions.
4. Supported directly by the programming language.
5. Not supported with Pascal.

6. Use the C language library subroutines SETJMP( ) to establish a label and LONGJMP( ) to perform the nonlocal GOTO back to that label. For more information, see the *C User's Guide*.

When you create on-units, remember that

- On-units can pass control in one of three ways: by calling another procedure, by using a local or nonlocal GOTO, or by returning to the calling procedure. They may not call EXIT, though they may GOTO a point in the main program which does so.
- On-units may set error codes as return parameters, print error messages, or signal other error conditions. They may not call ERRRTN or use ER$PRINT with any key other than the immediate-return key (K$IRTN).
- Programs containing on-units must be compiled in either V mode or I mode.
- Procedures that are on-units must take at least one argument.

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. Appendix A describes the system-defined conditions.

Scope of On-units

On-units are usually defined at the beginning of a program or subroutine, but they can be defined at any point within the program. When the program reaches the point at which the on-unit is defined (that is, a call to MKONU$, MKON$P, or MKON$F), the on-unit is said to be set. However, the on-unit does not execute unless the condition to which it responds is raised. An on-unit remains set until one of three things happens:

- The procedure within which the on-unit was defined returns (ends).
- A new on-unit for the condition is defined.
- The on-unit is reverted (disabled) by a call to RVONU$ or RVON$F.

Thus, if an on-unit for the condition ARITH$ is defined at the beginning of a program, it remains in effect throughout the program, unless it is reverted or some other on-unit for ARITH$ is defined later in the program. If a subroutine within that program defines its own on-unit for ARITH$, then that on-unit takes precedence (but only while the subroutine is executing). Each call to the subroutine reestablishes its on-unit; each return from the subroutine reverts the new on-unit and reestablishes the on-unit defined in the main program. (If no

on-unit is defined within the main program, then PRIMOS on-units are in effect when the main program is running.)

FORTRAN Considerations

The use of on-units and of nonlocal GOTOs is somewhat restricted in FORTRAN, because 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 because 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 *Subroutines Reference I: Using Subroutines.*

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* in the Data Structure Formats section 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 PL/I */

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 PL/I */
/* 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 GOTO.

Two Protecting Programs in CPL

Below are two programs, each of which protects a FORTRAN program called SQRT against being interrupted by the BREAK (or CONTROL–P) key. They demonstrate both a simple and a more sophisticated means by which programs can avoid having to use the condition mechanism subroutines. When the language in which a program 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
C  be protected from BREAKs (the QUIT$ condition) by an
C  enveloping 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$
C  condition.  If the user breaks out of the program
C  during its operation, and then reenters it through
C  the PRIMOS REN command, the on-unit is invoked to
C  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.
C  The on-unit prevents BREAK from exiting the program
C  and instructs the user how to exit.
C
C  In FTN the on-unit must be declared as an external
C  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
C  raised. Ordinarily, it would be incorrect to use
C  FORTRAN I/O from inside this on-unit, because FTN is
C  not reentrant, and we would be disturbing the keyboard

C  I/O that was in progress when QUIT$ was raised.  In
C  this case, however, we use a nonlocal GOTO to return
C  to statement 1000 of the main program, and never
C  return 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
C  CLEANUP$ 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
C  is 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$	Continue scan for on-units.
MKLB$F	Convert FORTRAN statement label to PL/I format.
MKON$F	Create an on-unit (for FTN users).
MKON$P	Create an on-unit (for any language except FTN).
MKONU$	Create an on-unit (for PMA and PL/I users).
PL1$NL	Perform a nonlocal GOTO.
RVON$F	Revert an on-unit (for FTN users).
RVONU$	Revert an on-unit (for any language except FTN).
SGNL$F	Signal a condition (for FTN users).
SIGNL$	Signal a condition (for any language except FTN).

CNSIG$

CNSIG$ is called when an on-unit has been unable to handle the condition completely. 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

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

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, cnamel, unit*)

Parameters

cname

INPUT. Array containing name of condition for which on-unit is to be created.

cnamel

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 *cnamel* 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

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

The PL/I usage is

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$

PL/I and PMA programmers can call MKONU$ to create an on-unit for a specific condition or a default on-unit for the ANY$ condition.

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 active on-unit for this condition is 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.

Caution PMA and PL/I are the only two languages you can use to call MKONU$. FORTRAN 77 programmers must use MKON$P and FORTRAN (FTN) programmers must use MKON$F. PL/I programmers can use either MKON$P or MKONU$.

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

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

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$

RVONU$ disables (reverts) an on-unit for a specific condition for any language except FTN.

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

Once disabled, an 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.

A call to RVONU$ has 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

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 (:1777600000).

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$

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$CLR	Disable signalling of EXIT$ condition.
EX$RD	Return state of EXIT$ signalling.
EX$SET	Enable signalling of EXIT$ condition.

EX$CLR

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

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

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 the Crawlout Mechanism section 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.

Semaphores and Timers

8

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 specific timed intervals. The third part describes routines that suspend (sleep) a process for a specified interval.

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 two or more processes. Associated timer subroutines allow you to wait a process on a semaphore for a specified interval or until notified.

Note Another method of coordinating two or more active processes is to use event synchronizers and their associated timers. Although they perform many of the same operations, event synchronizers and semaphores are independent and fundamentally different facilities. Synchronizers are user resources. Semaphores are shared system resources. Synchronizers are described in *Subroutines Reference V: Event Synchronization.*

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 two or more processes with one another. Such coordination is required when two or more processes cooperate to solve a common problem, or when two or more processes must use a common, limited resource.

When two or more 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).

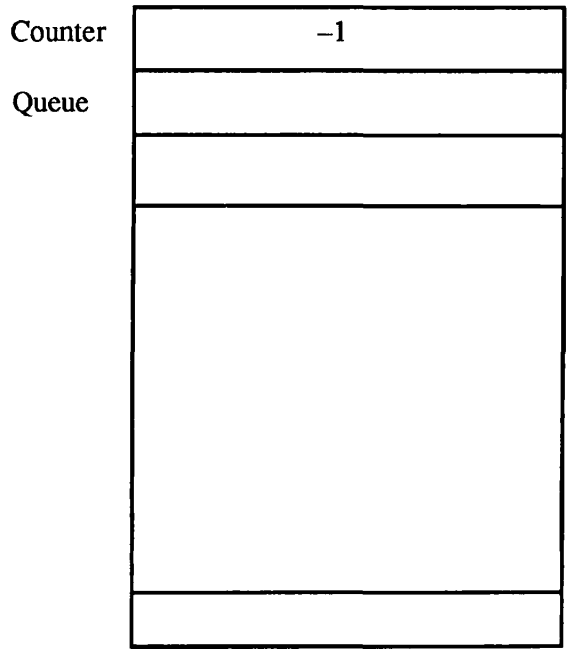


Figure 8–1. Resource Semaphore at Start

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).

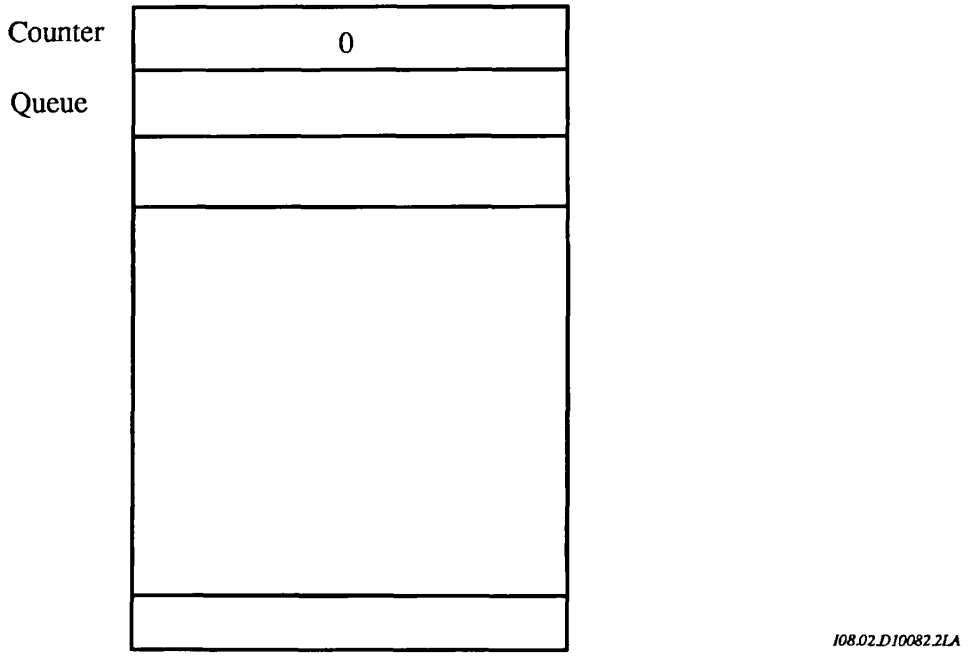


Figure 8–2. Resource Semaphore After Call by One Process (Process 1 is Using the Resource, No Processes Waiting)

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.

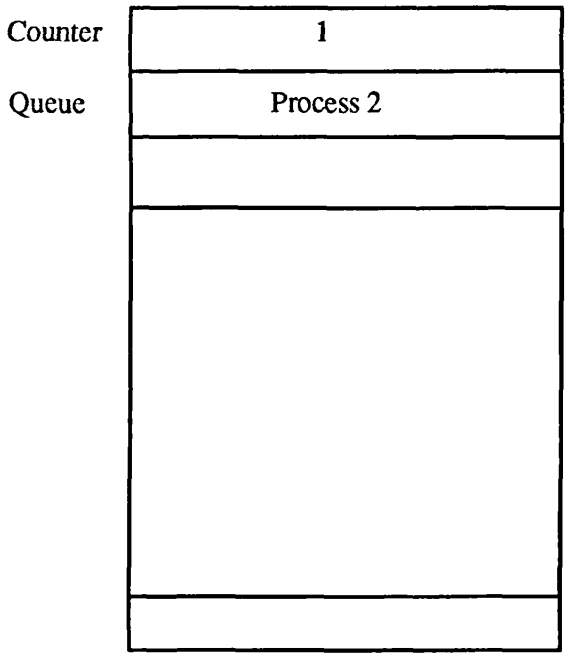


Figure 8–3. Resource Semaphore After Call by Second Process (First Process is Using the Resource)

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.

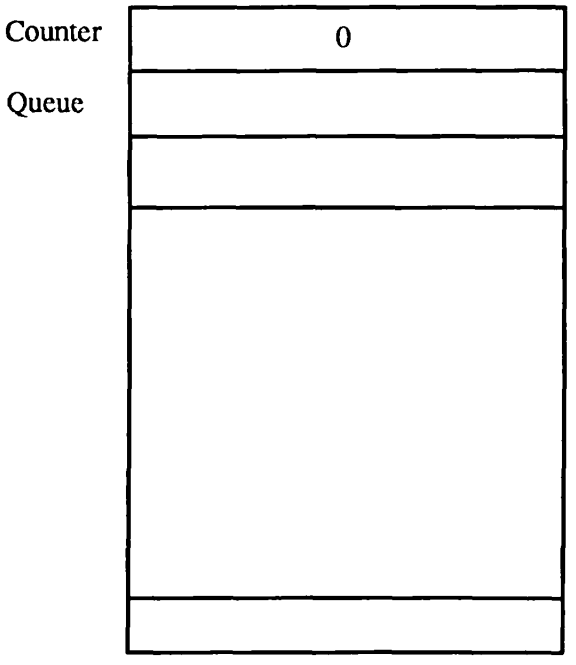


Figure 8–4. Resource Semaphore After Notify by One Process (Process 2 is Now Using the Resource)

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 through an open routine. The user can request that two or more 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 Versus Named Semaphores

The operating system maintains two different sets of semaphores, and processes must access these sets by different methods. 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 how each numbered semaphore is to be used; 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$OP will open a set of named semaphores and associate them with the name of a file in the current directory of the process performing the open operation. SEM$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, through 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.

Note that if another method is not used besides the semaphore to indicate that the awaited event has actually occurred, 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 why 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 the Coding Suggestion section, 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 reissue the wait call.

Infinite Waits

It is possible to create a situation in which one or more processes are waiting on a semaphore, and there are no processes running that will ever notify that semaphore. The following are methods of creating this situation.

Multiple Waits: If a process issues a wait call, and is *not* queued, and then continues to reissue the wait call without intervening notifies, that process will

eventually cause the semaphore count to become greater than 0 and the process will wait. This of course assumes that there is not another process somewhere doing multiple notifies.

In the case of a resource-protection semaphore, if all other processes obey the rules, they will wait on this semaphore before they notify it. They will therefore queue up behind the multiple-waiter process. Eventually, all the processes of the subsystem will become queued on the semaphore queue, and no process will remain to notify the semaphore.

Aborted Notifiers: Another way of causing infinite waits is to abort a process that would, under normal circumstances, notify a semaphore. If this is the only process that will do notifies on the semaphore, then all other processes that wait on it will wait forever.

Coding Suggestion: Infinite waits can be avoided by associating a timer with the semaphore. This will guarantee that one or more processes will eventually be removed from the wait queue. Extra coding must be done in the processes, however, so that a timeout can be identified as such, and so that appropriate action can be taken. This code should determine whether the process that should have notified the semaphore is still running or not. If it is running, the notify is considered external and the process reissues the wait call. If the potential notifiers have all been aborted, appropriate recovery action should be initiated.

Deadly Embrace

When two or more 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 use 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 databases. 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 two or more 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 two or more 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$CL	Release (close) a named semaphore.
SEM$DR	Drain a semaphore.
SEM$NF	Notify a semaphore.
SEM$OP	Open a set of named semaphores.
SEM$OU	Open a set of named semaphores.
SEM$TN	Periodically notify a semaphore.
SEM$TS	Return number of processes waiting on a semaphore.
SEM$TW	Wait on a specified named semaphore, with timeout.
SEM$WT	Wait on a semaphore.

SEM$CL

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

E$OK	Success.
E$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

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

E$OK	Success.
E$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$DR 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

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

E$OK	Success.
E$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

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

E$OK	Success.
E$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$OP opens a set of named semaphores associated with the name of a file in the current directory of the process performing the open operation. If the process has at least read-access rights to the file, it will be assigned the semaphores. Each semaphore is initialized to zero. SEM$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 zero 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 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 through 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 zero 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 through 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

This routine causes PRIMOS to notify a specified numbered semaphore after a specified interval of time, and periodically thereafter if so desired.

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 range 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 time, and subsequent times. If *int2* is 0, then the semaphore will be notified only once: after *int1* milliseconds.

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

E$OK	Success.
E$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 15 such timers per system.

The time intervals, quoted in milliseconds, are truncated to the nearest tenth of a second before being used.

When a call to SEM$TN specifies a semaphore that already has an active timer request, the values of *int1* and *int2* specified in the second call overwrite the values stored in the active timer.

Note SEM$TN sets a timer that increments a system semaphore. Unless halted, this timer continues to increment the semaphore after your program has completed. The semaphore does not halt until the timer has posted 32767 notices, which may take hours or days. To prevent this unnecessary system overhead, the last call to SEM$TN for each semaphore used in your program should specify 0 for both time arguments, *int1* and *int2*. This halts the timer. Use the LIST_SEMAPHORES command to determine the current number of notices posted to system semaphores.

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

This function returns the number of processes waiting on the specified 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

E$OK	Success.
E$BPAR	An invalid value was supplied for *snbr*.

Discussion

SEM$TS tests the semaphore's counter for the number of processes waiting on the semaphore. 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

This routine waits the process on the specified semaphore until it is either taken off the wait queue by a notify, or a specified amount of time 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, intl, 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.

intl

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. (*Do not* check for E$OK.) 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 *intl*.
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

This routine waits the process on the specified semaphore until it is taken off the wait queue by a notify.

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

E$OK	Success.
E$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 earlier in this chapter.

Loading and Linking Information

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

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

R-mode: No special action.

Limit Timer Routine

This section describes the following subroutine:

Routine	*Function*
LIMIT$	Set and read various timers.

LIMIT$

This routine sets or reads various timers within PRIMOS for the current process.

Usage

```
DCL LIMIT$ ENTRY (FIXED BIN(15), FIXED BIN(31),
                  FIXED BIN(15), FIXED BIN(15));

CALL LIMIT$ (key, limit, res, code);
```

Parameters

key

INPUT. *key* is split into two 8-bit functions. The right half (lower byte) specifies whether to read or set the timer limit. Specify one of the following values:

1	Read the limit.
2	Set the limit.

The left half (upper byte) of *key* specifies which time limit to either read or set. Each limit performs a different action when its value decrements to zero. Unless otherwise specified, these timers are set by a call to LIMIT$ and decremented by PRIMOS. Specify one of the following values:

1	CPU login limit, in seconds. PRIMOS logs the user out when the user has consumed this number of seconds of CPU time. LIMIT$ can read or modify this limit; the timer for this limit is set to the system default and begins to decrement when the user logs in. You cannot set the CPU login limit larger than 1000000 (one million seconds). You cannot set the CPU login limit to a value larger than its current reading; once set, this value may only be decreased. Setting the CPU login limit to zero logs the user out immediately.

2 Realtime login limit, in minutes. PRIMOS logs the user out when the number of minutes specified in *limit* has elapsed. Reading the realtime login limit returns the number of realtime minutes remaining before the user is logged out. Setting the realtime login limit is done by setting the *limit* parameter to the number of minutes. The realtime login limit should not be set larger than 32768 minutes; doing so may produce unpredictable results. To cancel the realtime login limit, set *limit* to zero.

5 CPU watchdog limit, in seconds. PRIMOS raises the CPU_TIMER$ condition when the number of seconds specified in *limit* has elapsed. Reading the CPU watchdog limit returns the number of CPU seconds remaining before the CPU_TIMER$ condition will be raised. The CPU watchdog limit cannot be set larger than 1000000 (one million seconds). To cancel the CPU watchdog limit, set *limit* to zero.

6 Realtime watchdog limit, in minutes. PRIMOS raises the ALARM$ condition when the number of minutes specified in *limit* has elapsed. Reading the realtime watchdog limit returns the number of realtime minutes remaining before the ALARM$ condition will be raised. Setting the realtime watchdog limit to zero cancels the limit.

7 Realtime watchdog limit, in seconds. PRIMOS raises the ALARM$ condition when the number of seconds specified in *limit* has elapsed. Reading the realtime watchdog limit returns the number of realtime seconds remaining before the ALARM$ condition will be raised. Setting the realtime watchdog limit to zero cancels the limit.

8 Inactivity logout limit, in minutes. PRIMOS logs the user out when the process has remained idle for the number of minutes specified in *limit*. Reading the inactivity logout limit returns the number of minutes the process must remain idle before the user will be logged out. The inactivity logout limit cannot be set to zero or increased beyond its current setting.

limit

INPUT. The time limit to be set, in minutes or seconds. This value cannot be less than zero.

OUTPUT. The time limit to read, in minutes or seconds. Reading a value of zero means the limit is not active.

res

INPUT. Reserved — must be zero.

code

OUTPUT. Standard error code. Possible values are

E$OK	No error.
E$BKEY	Invalid value specified in *key*.
E$BPAR	Invalid value specified for *limit*, or nonzero value specified for *res*.

Discussion

LIMIT$ sets and reads timers within PRIMOS, and performs a different action when each type of timer expires. These actions include logging out the user and raising standard conditions. For information about particular timers, see the description of the *key* parameter above.

Once the CPU login limit has been set, the user may only call LIMIT$ to specify a shorter CPU login limit. The user cannot use LIMIT$ to specify a longer CPU login limit than the current setting (time remaining) or to cancel the CPU login limit.

The realtime watchdog limit, in minutes, and the realtime watchdog limit, in seconds, are the same limit. When read, all limits expressed in minutes are rounded up to the next highest minute. Therefore, nonzero limits of less than one minute are read as one minute. Fractions of a second are rounded up to the next highest second.

For each user, the inactivity logout limit is set to the system default when the user logs in. Thereafter, the user may only call LIMIT$ to specify a shorter inactivity logout limit than the its current setting. The user cannot use LIMIT$ to specify a longer inactivity logout limit than the current setting or to cancel the inactivity logout limit.

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$	Suspend a process for a specified interval.
SLEP$I	Suspend a process for a specified interval (interruptible).

SLEEP$

This routine suspends the process for a specified interval. Not interruptible.

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

This routine suspends the process for a specified interval. Interruptible.

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.

Message Facility

9

The PRIMOS message facility includes calls for sending and receiving interuser messages.

These interuser messaging subroutines can also set and query a user's willingness to receive messages. These 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.

Note User processes can also exchange messages using the InterServer Communication facility (ISC). ISC is described in *Subroutines Reference V: Event Synchronization.*

Message Facility Routines

This section describes the following subroutines:

Routine	*Function*
MSG$ST	Return the receiving state of a user.
MGSET$	Set the receiving state for messages.
RMSGD$	Receive a deferred message.
SMSG$	Send an interuser message.

MSG$ST

This routine enables the caller to determine whether a process is set to accept, defer, or reject messages sent to the process via the PRIMOS message facility.

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 name and receive state for process *user_num* on system *system_name*.
2	Return the user number and receive state for process *user_name* on system *system_name*.

user_num

INPUT or OUTPUT. The user number of the process. If *key* = K$READ, the caller must supply *user_num* as an input argument. If *key* = 2, the subroutine returns *user_num*. If no process by the name *user_name* is logged in, MSG$ST returns zero to *user_num*.

system_name

INPUT. The name of the system on which the desired process is to be search for.

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, the subroutine returns *user_name*. If *key* = 2, the caller must specify *user_name* as an input parameter.

user_name_len

INPUT. The length of *user_name* in characters. The maximum length is 32 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

MSG$ST enables the caller to determine whether a process is set to accept, defer, or reject messages sent via the PRIMOS message facility. This setting is referred to as the receive state of the process. In the *key* argument, the caller can choose to specify a process either by its user name or user number.

- If *key* = K$READ, the caller must supply the user number of the process as an input parameter in *user_num*. MSG$ST returns the name and receive state of this process.
- If *key* = 2, the caller must supply the user name of the process as an input parameter in *user_name*. MSG$ST returns the user number and receive state of the most permissive process by that user name. If there is more than one such process, MSG$ST returns the lowest user number among them. Note: a process set to accept messages is more permissive than a process set to defer them; a process set to defer messages is more permissive than a process set to reject them.

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$

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$OK	No error.
E$BKEY	Bad key.

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$

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$

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$OK	Operation succeeded.
E$NRCV	Operation aborted because send does not have receive enabled.
E$UADR	Unknown addressee.
E$PRTL	Operation partially blocked.
E$NSUC	Operation failed.

error_vector(2) — Three less than the total number of elements in *error_vector*. This value is provided by the user, and is normally set to the number of configured users. At Rev. 22.0 and subsequent revisions, this number can be as high as 991; for previous revisions, it can be as high as 255.

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*(4) on. If you set *error_vector*(2) greater than KUSR, SMSG$ will only use the number of elements specified in KUSR. Thus, if you are not interested in the information, this large buffer need not be reserved. However, you must allocate an *error_vector* array large enough to contain the *error_vector*(2) that you specified, plus 3 words.

---

error_vector(4–994)	An optional status vector whose maximum length is either the value of *error_vector*(2) + 3, or KUSR + 3, whichever is smaller. If you supply this optional vector, SMSG$ returns to each element a status code that indicates 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 of *key* 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.

Standard Conditions

A

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
       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 in the Data Structure Formats section. 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 PL/I 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;
```

target_label

Label to which the nonlocal GOTO was attempted.

ptr_to_nlg_call

Pointer to the call to PL1$NL that requested this nonlocal GOTO.

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.

BAD_RECORD_ADDRESS$

(software, not returnable)

An internal inconsistency has been detected when trying to read a block of a file. No further attempt should be made to access the file, and the System Administrator should be informed.

Information Structure:

```
DCL 1  info BASED,
       2  pathname CHAR(128) VAR,
       2  ldev FIXED BIN,
       2  ra FIXED BIN(31),
       2  bra FIXED BIN(31);
```

pathname

The name of the file having the problem. The string `*** unavailable ***` is returned if PRIMOS cannot retrieve the name.

ldev

The logical device number of the partition containing the file.

ra

The Record Address at which the error occurred. This may be reported to the System Administrator.

bra

The Branch Record Address of the file. This may be reported to the System Administrator.

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: The standard PL/I condition information structure is provided.

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.

DAMAGED_RAT$

(software, not returnable)

PRIMOS has detected an inconsistency in the Record Availability Table of a disk partition. The System Administrator should be informed.

Information Structure:

```
DCL 1  info,
       2  ldev FIXED BIN;
```

ldev

Logical device number of the partition on which the error occurred.

DISK_READ_ERR$

(software, not returnable)

A nonrecoverable error has occurred when trying to read a file. No further attempt should be made to access the file, and the System Administrator should be informed.

Information Structure:

```
DCL 1  info BASED,
       2  pathname CHAR (128) VAR,
       2  ldev FIXED BIN,
       2  ra FIXED BIN(31),
       2  bra FIXED BIN(31);
```

pathname

The name of the file having the problem. The string `*** unavailable ***` is returned if PRIMOS cannot retrieve the name.

ldev

The logical device number of the partition containing the file.

ra

The Record Address at which the error occurred. This may be reported to the System Administrator.

bra

The Branch Record Address of the file. This may be reported to the System Administrator.

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.

Information Structure: 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.

Information Structure: 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-specific 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 ER$PRINT 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.

Information Structure: 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, Memory Allocation.

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: A condition frame header (CFH) in the standard format describing 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.

INVALID_REC_ADR$

(software, not returnable)

PRIMOS attempted to free a disk record whose address is invalid. This indicates an inconsistency on the disk partition. The System Administrator should be informed.

Information Structure:

```
DCL 1 info,
      2  ldev FIXED BIN;
```

ldev
Logical device number of the partition on which the error occurred.

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.

LIBRARY_IO_ERR$

(software)

This condition indicates that the FORTRAN I/O library has encountered an unexpected error. The condition may be returnable or not, depending on the circumstances. Examine the Condition Frame Header (the field is *cflags.return_ok*) to find out if you can return from the condition.

The default on-unit prints a diagnostic message on the terminal and signals the STOP$ condition.

Information Structure:

```
DCL 1  info BASED,
       2  error_code FIXED BIN,
       2  logical_unit FIXED BIN,
       2  routine_name CHAR(32) VAR,
       2  error_message CHAR(80) VAR;
```

error_code

Standard error code defining the error encountered.

logical_unit

Number of the FORTRAN logical unit on which the error was encountered.

routine_name

Name of the routine that encountered the error.

error_message

Additional information about the error.

LINKAGE_ERROR$

(hardware, returnable)

The process made a reference through an indirect pointer (IP) that is a valid unsnapped dynamic link. Either an error occurred while attempting to resolve the fault, or an attempt was made to create an invalid dynamic link type. Process-class library EPFs are prevented from linking to either program-class library EPFs or static-mode shared libraries.

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,
      2  entry_name_ptr POINTER OPTIONS(SHORT),
      2  errcode FIXED BIN;
```

entry_name_ptr

Pointer to the name of the entry point that could not be found.

errcode

Standard error code.

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;
```

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 BIN;
/* 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 reading a stream assignment in a GET statement 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.

NAMELIST_LIB_ERR$

(software, returnable)

This condition is used by the subroutines in the F77 NAMELIST library to report to the user additional information concerning an error.The information is stored in a user buffer whose structure is given below.

Information Structure:

```
DCL 1 namelist_err_info BASED,
       2 code FIXED BIN,
       2 routine_name CHAR(32) VAR,
       2 namelist_name CHAR(6) VAR,
       2 input_line_ptr PTR OPTIONS(SHORT),
       2 line_err_ptr FIXED BIN,
       2 input_line_length FIXED BIN;
```

code

Standard error code defining the error encountered.

routine_name

Name of the NAMELIST library subroutine that encountered the error.

namelist_name

Name given to the buffer in the F77 read or write.

input_line_ptr

Pointer to the buffer.

line_err_ptr

Offset into the buffer of the information about the error.

input_line_length

Length of the buffer.

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 BIN,
          /* npx node number on which slave is running */
      2  orig_condition CHAR(32) VAR,
          /* condition raised in slave */
      2  orig_info_data (129) FIXED BIN;
           /* 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.

PAGING_DEVICE_FULL$

(hardware, returnable)

The process encountered a page fault referencing a valid virtual address, but the page had not previously been assigned room on the paging disk, and no more room was available. If the on-unit for this condition returns, the reference is retried, with some likelihood that a page has been made available by another process.

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 allocated.

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.

RTNREC_ERR$

(software, not returnable)

PRIMOS has attempted to free a disk record that is already marked as free. This indicates an inconsistency on the disk partition. The System Administrator should be informed.

Information Structure:

```
DCL 1  info,
       2  ldev FIXED BIN;
```

ldev
Logical device number of the partition on which the error occurred.

R0_ERR$

(software, returnable)

A ring-0 call to ER$PRINT 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.

SETRC$

(software, returnable)

This condition is raised when a program that called subroutine SETRC$ exits. See the discussion in Chapter 5 for information about the SETRC$ subroutine.

No information structure is available.

The default on-unit simply returns.

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.

Information Structure: 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 PL/I, the condition can be raised either through the ALLOCATE statement or by the compiler making its own call.

Information Structure: 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: The standard PL/I condition information structure is provided.

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;
```

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 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.

Information Structure: 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 raised when an attempt is made to divide a floating-point or fixed-point number by zero. (Note that the ZERODIVIDE condition is not raised when an attempt is made to divide an integer by zero.) 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.

By default, a bit in the ECB keys register is set to OFF to disable the ZERODIVIDE condition. The condition can be enabled by writing a PMA program to set the bit ON.

Information Structure: The standard PL/I condition information structure is provided.

Data Type Equivalents

B

To call a subroutine from a program written in any Prime language, you must declare the subroutine and its parameters in the calling program. Therefore, you must translate the PL/I data types expected by the subroutine into the equivalent data types in the language of the calling program.

Table B-1 shows the equivalent data types for the Prime languages BASIC/VM, C, COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, and PL/I. The leftmost column lists the generic storage unit, which is measured in bits, bytes, or halfwords for each data type. Each storage unit matches the data types listed to the right on the same row. The table does not include an equivalent data type for each generic unit in all languages. However, with knowledge of the corresponding machine representation, you can often determine a suitable workaround. For instance, to see if you can use a left-aligned bit in COBOL 74, you could write a program to test the sign of the 16-bit field declared as COMP. In addition, if a subroutine parameter consists of a structure with elements declared as BIT(*n*), it can be declared as an integer in the calling program. Read the appropriate language chapter in the *Subroutines Reference I: Using Subroutines* before using any of the equivalents shown in the table.

Note The term PL/I refers both to full PL/I and to PL/I Subset G (PL/I-G).

Table B-1. Equivalent Data Types for Prime Languages

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
16-bit integer	INT	short enum	COMP PIC S9(1)- PIC S9(4)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER Enumerated	FIXED BIN FIXED BIN(15)
32-bit integer	INT*4	int long	COMP PIC S9(5)- PIC S9(9)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	FIXED BIN(31)
64-bit integer			COMP PIC S9(10)- PIC S9(18)				
32-bit float single precision	REAL	float	COMP-1	REAL REAL*4	REAL REAL*4	REAL	FLOAT BIN FLOAT BIN(23)
64-bit float double precision	REAL*8	double	COMP-2	REAL*8	REAL*8	LONGREAL	FLOAT BIN(47)
128-bit float quad precision					REAL*16		
1 bit		short					BIT BIT(1)
1 left-aligned bit		short				BOOLEAN	BIT(1) ALIGNED

T0F01.D10082.2LA

Table B-1. Equivalent Data Types for Prime Languages (Continued)

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
Bit string		unsigned int				SET	BIT(n)
Fixed-length character string	INT	char NAME[n] char NAME	DISPLAY PIC A(n) PIC X(n) FILLER		CHARACTER *n	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)
Fixed-length digit string			DISPLAY PIC 9(n)				PICTURE
Fixed-length digit string, 2 digits per byte			COMP-3				FIXED DECIMAL
Varying-length character string		struct {short LENGTH; char DATA[n]; } CVAR				STRING[n]	CHAR(n) VARYING
32-bit pointer		Pointer (32IX-mode)		LOC( )	LOC( )		POINTER OPTIONS (SHORT)
48-bit pointer		Pointer (64V-mode)				Pointer	POINTER

T0F01.D10082.2LA

Notes

For a discussion of possible workarounds for some of the empty boxes in this table as well as a description of generic units for PMA, refer to the appropriate language chapter in the *Subroutines Reference I: Using Subroutines.*

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

File-system Date Format

C

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.

Superseded Routines

D

This appendix lists routines considered obsolete or superseded, which Prime continues to support. It describes the following subroutines:

Routine	*Function*
DISPLY	Update sense light settings.
ERRPR$	Print a standard error message.
ERRSET	Set ERRVEC (a system error vector).
ERTXT$	Return text representation of error code.
GETERR	Return ERRVEC contents.
OVERFL	Check if an overflow condition has occurred.
PHANT$	Start a phantom process.
PRERR	Print an error message.
RECYCL	Cycle to the next user.
SLITE	Set the sense light on or off.
SLITET	Test sense light settings.
SSWTCH	Test sense switch settings.
TEXTO$	Check filename for valid format.
UPDATE	Update current directory (PRIMOS II only).

DISPLY

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.

ERRPR$

ERRPR$ interprets a return code and, if the code is non zero, prints a standard message associated with the code, followed by optional user text. See *Subroutines Reference I: Using Subroutines* 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.

ERRSET

ERRSET sets ERRVEC, a system vector, then takes an alternate return or prints the message stored in ERRVEC and returns control to the system.

Usage

CALL ERRSET (*altval, altrtn*)

CALL ERRSET (*altval, altrtn, messag, num*)

CALL ERRSET (*altval, altrtn, name, messag, num*)

Parameters

In Form 1, *altval* must have value 100000 octal and *altrtn* specifies where control is to pass. If *altrtn* is zero, 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 zero, 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 zero.

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 zero, 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.

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: Not available.

GETERR

This routine returns the contents of ERRVEC.

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

This routine checks if an overflow condition has occurred.

Usage

CALL OVERFL (A1)

Discussion

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.

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$

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*)

Parameters

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 E$OK, the phantom was initiated successfully. If *code* is E$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

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.
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.
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	Directory 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.

RECYCL

RECYCL tells PRIMOS to cycle to the next user. It is an I-have-nothing-to-do-for-now call. Under PRIMOS II, RECYCL does nothing.

Usage

```
CALL RECYCL
```

Caution Do not use RECYCL to simulate a time delay.

SLITE

This routine 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

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

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$

TEXTO$ checks a filename for valid format. This subroutine has been replaced with FNCHK$.

Usage

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

Parameters

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

Under PRIMOS II, this subroutine updates the current directory.

Usage

CALL UPDATE (*key*, 0)

Parameters

key

Value must be 1 to update current directory, 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.

Index of Subroutines by Function

This index lists subroutines grouped by the general functions that they perform. See the Index of Subroutines by Name to find a particular subroutine's volume, chapter, and page number.

Access Category

Add an object's name to an access category.	AC$CAT
Modify an existing ACL on an object.	AC$CHG
Set an object's ACL to that of its parent directory.	AC$DFT
Make an object's ACL identical to that of another object.	AC$LIK
Obtain the contents of an object's ACL.	AC$LST
Convert an object from ACL protection to password protection.	AC$RVT
Set a specific ACL on an object.	AC$SET
Determine whether an object is accessible for a given action.	CALAC$
Delete an access category.	CAT$DL
Obtain the user-ID and the groups to which it belongs.	GETID$
Obtain the passwords of a subdirectory of the current directory.	GPAS$$
Determine whether an object is ACL-protected.	ISACL$
Remove an object's priority access.	PA$DEL
Obtain the contents of an object's priority ACL.	PA$LST
Set priority access on an object.	PA$SET
Set the owner and nonowner passwords on an object.	SPAS$$

Access Server Names

Catalog a server's Low Level Name.	ISN$C
Look up a server's Low Level Name.	ISN$L
Recatalog a server's Low Level Name.	ISN$RC
Uncatalog a server's Low Level Name.	ISN$UC
Get the server name of a process.	SRS$GN

Get the process numbers of all processes associated with the server name.	SRS$GP
List the server names on your system.	SRS$LN

Arrays

Get a character from an array.	GCHAR
Store a character into an array location.	SCHAR

Asynchronous Lines

Return asynchronous line characteristics.	AS$LST
Return an asynchronous line number.	AS$LIN
Set asynchronous line characteristics.	AS$SET

Attach Points

Set the attach point to a directory specified by the pathname.	AT$
Set the attach point to a specified top-level directory and partition.	AT$ABS
Set the attach point to a specified top-level directory on any partition.	AT$ANY
Set the attach point to the home directory.	AT$HOM
Set the attach point to a specified top-level directory on a partition identified by logical disk number.	AT$LDEV
Set the attach point to the login directory.	AT$OR

Set the attach point to a directory subordinate to the current directory.	AT$REL
Set the attach point to the root directory.	AT$ROOT
Set the attach point to a specified directory, and optionally, make it the home directory.	ATCH$$

Binary Search

Perform binary search in ordered table.	BIN$SR

Buffer Output

Provide free-format output to a buffer.	IOA$RS

Command Environment

Return caller's maximum command environment breadth.	CE$BRD
Return caller's maximum command environment depth.	CE$DPT
Parse command arguments according to a character string "picture" of the command line.	CL$PIX
Invoke a command from a running program.	CP$
Retrieve the value of a global variable.	GV$GET
Set the value of a global variable.	GV$SET
Return a list of commands valid at mini-command level.	LIST$CMD
Retrieve the value of a CPL local variable.	LV$GET

Set the value of a CPL local variable.	LV$SET
Return breadth of caller's current command environment.	RD$CE_DP

Command Level

Call a new command level after an error.	CMLV$E
Call a new command level.	COMLV$
Return to PRIMOS.	EXIT
Initialize the command environment.	ICE$
Return serialization data.	KLM$IF
Record command error status.	SETRC$
Signal an error in a subsystem.	SS$ERR

Condition Mechanism

Continue scan for on-units.	CNSIG$
Convert FORTRAN statement label to PL/I format.	MKLB$F
Create an on-unit (for FTN users).	MKON$F
Create an on-unit (for any language except FTN).	MKON$P
Create an on-unit (for PMA and PL/I users).	MKONU$
Perform a nonlocal GOTO.	PL1$NL
Revert an on-unit (for FTN users).	RVON$F
Revert an on-unit (for any language except FTN).	RVONU$
Signal a condition (for FTN users).	SGNL$F
Signal a condition (for any language except FTN.)	SIGNL$

Controllers, Asynchronous, Multi-line

Communicate with SMLC driver.	T$SLC0
Assign AMLC line.	ASNLN$
Communicate with AMLC driver.	T$AMLC

Data Conversion

Convert a string from lowercase to uppercase or uppercase to lowercase.	CASE$A
Convert ASCII number to binary.	CNVA$A
Convert binary number to ASCII.	CNVB$A
Make a number printable if possible.	ENCD$A
Convert the DATMOD field (as returned by RDEN$$) in format DAY, MON DD YYYY	FDAT$A
Convert the DATMOD field (as returned by RDEN$$) in format DAY, DD MON YYYY.	FEDT$A
Convert the TIMMOD field (as returned by RDEN$A).	FTIM$A

Date Formats

Convert binary date to quadseconds.	CV$DQS
Convert ASCII date to binary format.	CV$DTB
Convert binary date to ISO format.	CV$FDA
Convert binary date to visual format.	CV$FDV
Convert quadsecond date to binary format.	CV$QSD

Devices, Assigning or Attaching

Attach specified devices.	ATTDEV
Provide or set aside available logical file unit.	IOCS$G
Free a logical file unit number.	IOCS$F

Disk I/O

Read ASCII from disk.	I$AD07
Write binary to disk.	O$BD07
Read binary from disk.	I$BD07
Write ASCII to disk (fixed-length records).	O$AD08
Register disk format with driver.	DKGEO$

Drivers, Device-independent

Write ASCII data.	WRASC
Read ASCII data.	RDASC
Write binary data.	WRBIN
Read binary data.	RDBIN
Open PRIMOS file and perform other nondata transfer functions. (Primarily for IOCS applications.)	CONTRL

Encryption, of Login Password

Encrypt login validation passwords.	ENCRYPT$

EPFs

Allocating and Deallocating Space for EPFs

Allocate space for EPF function return information.	ALC$RA
Allocate space and set value of EPF function return information.	ALS$RA
Deallocate space for EPF function return information.	FRE$RA

Management of EPFs

Perform the linkage allocation phase for an EPF.	EPF$ALLC
Return the state of the command processing flags in an EPF.	EPF$CPF
Deactivate the most recent invocation of a specified EPF.	EPF$DEL
Perform the linkage initialization phase for an EPF.	EPF$INIT
Initiate the execution of a program EPF.	EPF$INVK
Map the procedure images of an EPF file into virtual memory.	EPF$MAP
Combine functions of EPF$ALLC, EPF$MAP, EPF$INIT, and EPF$INVK.	EPF$RUN
Modify user's search rules to allow dynamic linking to a library EPF.	LN$SET
Remove an EPF from a user's address space.	REMEPF$
Replace one EPF runfile with another.	RPL$

Information From In-memory User Profile

Return maximum number of dynamic segments.	DY$SGS
Return maximum number of static segments.	ST$SGS
Return highest segment number.	TL$SGS

Registering EPFs

Return ready or suspended status for registered EPF.	EPF$ISREADY
Enable registration of EPFs	EPF$REG
Enable unregistration of registered EPFs	EPF$UREG

Error Handling, I/O

Set ERRVEC and perform a return or display ERRVEC message before returning control to system.	ERRSET
Obtain contents of ERRVEC.	GETERR
Display I/O error message on user terminal.	PRERR

Event Synchronizers and Event Groups

Creating, Using, and Destroying Event Synchronizers

Create an event synchronizer.	SYN$CREA
Post a notice on an event synchronizer.	SYN$POST
Wait on an event synchronizer.	SYN$WAIT
Perform a timed wait on an event synchronizer.	SYN$TMWT

Retrieve a notice from an event synchronizer.	SYN$RTRV
Destroy an event synchronizer.	SYN$DEST

Creating, Using, and Destroying Event Groups

Create an event group.	SYN$GCRE
Move an event synchronizer into an event group.	SYN$MVTO
Remove an event synchronizer from an event group.	SYN$REMV
Cause a process to wait on an event group.	SYN$GWT
Cause a process to perform a timed wait on an event group.	SYN$GTWT
Retrieve a notice from an event group.	SYN$GRTR
Destroy an event group.	SYN$GDST

Getting Information About Synchronizers and Groups

Return number of notices or waiting processes on a synchronizer.	SYN$CHCK
Return number of notices on a group at one or all priority levels; if all levels, also return number of waiting processes.	SYN$GCHK
Indicate whether synchronizer is in group; and if it is, return the group number, priority level, and For Client Use field.	SYN$INFO
List the synchronizers in group and total number.	SYN$LSIG
List the synchronizers in server and total number.	SYN$LIST
List the groups in server and total number.	SYN$GLST

Executable Images

Restore an R-mode executable image.	REST$$
Restore and resume an R-mode executable image.	RESU$$
Save an R-mode executable image.	SAVE$$

EXIT$ Condition

Disable signalling of EXIT$ condition.	EX$CLR
Return state of EXIT$ signalling.	EX$RD
Enable signalling of EXIT$ condition.	EX$SET

File System Objects

Append a specified suffix to a pathname.	APSFX$
Extend or truncate a CAM file.	CF$EXT
Retrieve a CAM file's extent map from disk.	CF$REM
Set a CAM file's allocation size value.	CF$SME
Change the open mode of an open file.	CH$MOD
Close a file by name and return a bit string indicating closed units.	CL$FNR
Close a file system object by pathname.	CLO$FN
Close a file system object by file unit number.	CLO$FU
Close a file.	CLOS$A
Change the name of an object in the current directory.	CNAM$$
Create a new subdirectory in the current directory.	CREA$$
Create a new password directory.	CREPW$

Delete a file.	DELE$A
Create a new directory.	DIR$CR
Search for specified types of entries in a directory open on a file unit.	DIR$LS
Read sequentially the entries of a directory open on a file unit.	DIR$RD
Return entries meeting caller-specified selection criteria in a directory open on a file unit.	DIR$SE
Return the contents of a named entry in a directory open on a file unit.	ENT$RD
Generate a filename based on another name.	EQUAL$
Check for file existence.	EXST$A
Return a file system object's entryname and parent directory pathname.	EXTR$A
Delete a file identified by a pathname.	FIL$DL
Return information about a specified file unit.	FINFO$
Verify a supplied string as a valid filename.	FNCHK$
Force PRIMOS to write modified records to disk.	FORCEW
Position to end-of-file.	GEND$A
Return the pathname of a specified unit, attach point, or segment.	GPATH$
Tell whether the partition on which a file exists is robust.	GTROB$
Determine whether an open file system object is local or remote.	ISREM$
Return information on the system's list of logical disks.	LDISK$
List the disks a given user is using.	LUDSK$
Convert an existing directory entry to a portal by mounting the defined portal over the directory.	NAM$AD_PORTAL
Read the contents of the Global Mount Table; return a list of current-mounted disk partitions and the currently mounted portals accessible by the calling program	NAM$L_GMT
Remove a portal entry from the specified directory pathname.	NAM$RM_PORTAL

Open supplied name.	OPEN$A
Read name and open.	OPNP$A
Open supplied name with verification and delay.	OPNV$A
Read name and open with verification and delay.	OPVP$A
Return a logical value indicating whether a specified partition supports ACL protection and quotas.	PAR$RV
Position file.	POSN$A
Read, write, position, or truncate a file.	PRWF$$
Return directory quota and disk record usage information.	Q$READ
Set a quota on a subdirectory in the current directory.	Q$SET
Position in or read from a directory.	RDEN$$
Read a line of characters from an ASCII disk file.	RDLIN$
Return position of file.	RPOS$A
Rewind file.	RWND$A
Set or modify an object's attributes in its directory entry.	SATR$$
Delete a segment directory entry.	SGD$DL
Determine if a segment directory entry exists.	SGD$EX
Open a segment directory entry.	SGD$OP
Position in, read an entry in, or modify the size of a segment directory.	SGDR$$
Return the size of a file system entry.	SIZE$
Open, close, delete, change access, or verify the existence of an object.	SRCH$$
Search for a file with a list of possible suffixes.	SRSFX$
Open a scratch file with unique name.	TEMP$A
Verify a supplied string as a valid pathname.	TNCHK$
Truncate file.	TRNC$A
Scan the file system structure.	TSCN$A
Open a file anywhere in the PRIMOS file structure.	TSRC$$
Check for file open.	UNIT$A

Return the minimum and maximum file unit numbers currently in use by this user.	UNITS$
Return a logical value indicating whether a wildcard name was matched.	WILD$
Write a line of characters to a file in compressed ASCII format.	WTLIN$

ISC

Establish an ISC Session

Initiator requests the session.	IS$RS
Recipient gets the session request.	IS$GRQ
Recipient accepts the session.	IS$AS
Initiator gets the session request response.	IS$GRS

ISC Message Exchange

Allocate a buffer for a message data part.	IS$AB
Free an allocated data part buffer.	IS$FB
Send a message.	IS$SM
Receive a message.	IS$RM

Monitor ISC Message Exchange Session

Get sessions owned by your server.	IS$GSO
Get session attributes.	IS$GSA
Get session status.	IS$GSS
Get statistics about a session.	IS$STA

Terminate ISC Sessions or Respond to Exceptions

Terminate the caller's side of a session.	IS$TS
Get an exception.	IS$GE
Clear an exception.	IS$CE

Keyboard or ASR Reader

Input ASCII from terminal or ASR reader.	I$AA01
Perform same function as I$AA01 but also allow input from a cominput file.	I$AA12

Logging

Log a user message to the DMS server.	DS$SEND_CUSTOMER_UM

Matrix Operations

Generate permutations.	PERM
Generate combinations.	COMB

The following groups contain subroutines for single-precision, double-precision, integer, and complex operations, respectively.

(* indicates that a subroutine is not available.)

Set matrix to identity matrix.	MIDN, DMIDN, IMIDN, CMIDN
Set matrix to constant matrix.	MCON, DMCON, IMCON, CMCON
Multiply matrix by a scalar.	MSCL, DMSCL, IMSCL, CMSCL
Perform matrix addition.	MADD, DMADD, IMADD, CMADD
Perform matrix subtraction.	MSUB, DMSUB, IMSUB, CMSUB
Perform matrix multiplication.	MMLT, DMMLT, IMMLT, CMMLT
Calculate transpose matrix.	MTRN, DMTRN, IMTRN, CMTRN
Calculate adjoint matrix.	MADJ, DMADJ, IMADJ, CMADJ
Calculate inverted matrix.	MINV, DMINV, *, CMINV
Calculate signed cofactor.	MCOF, DMCOF, IMCOF, CMCOF
Calculate determinant.	MDET, DMDET, IMDET, CMDET
Solve a system of linear equations.	LINEQ, DLINEQ, *, CLINEQ

Memory

Allocate memory on the current stack.	ALOC$S
Move a block of memory.	MOVEW$
Make the last page of a segment available.	MM$MLP
Make the last page of a segment unavailable.	MM$MLP
Allocate user-class dynamic memory.	STR$AL
Allocate process-class dynamic memory.	STR$AP
Allocate subsystem-class dynamic memory.	STR$AS

Allocate user-class dynamic memory.	STR$AU
Free process-class dynamic memory.	STR$FP
Free user-class dynamic memory.	STR$FR
Free subsystem-class dynamic memory.	STR$FS
Free user-class dynamic memory.	STR$FU

Message Facility

Return the receiving state of a user.	MSG$ST
Set the receiving state for messages.	MGSET$
Receive a deferred message.	RMSGD$
Send an interuser message.	SMSG$

Numeric Conversions

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

Paper Tape

Control functions for paper tape.	C$P02
Input ASCII from the high-speed paper-tape reader.	I$AP02
Output binary data to the high-speed paper-tape punch.	O$BP02

Input one character from the high-speed paper-tape reader to Register A.	P1IB
Output one character to the high-speed paper-tape punch from Register A.	P1OB
Input one character from paper tape, set high-order bit, ignore line feeds, send a line feed when carriage return is read.	P1IN
Output one character to the high-speed paper-tape punch.	P1OU

Parsing

Parse a PRIMOS command line.	CMDL$A
Parse character string into tokens.	GT$PAR

Peripheral Devices

Line Printers

Centronics LP.	O$AL04
Parallel interface to line printer (MPC).	O$AL06
Versatec printer.	O$AL14
Move data to LPC line printer.	T$LMPC
Access a spooler queue.	SPOOL$
Place file in spool queue and perform SPOOLER command functions.	SP$REQ

Printer/Plotter

Versatec.	O$AL14
Versatec.	T$VG

Card Reader/Punch

Input from parallel card reader.	I$AC03
Input from serial card reader.	I$AC09
Read and print card from parallel interface reader.	I$AC15
Input from MPC card reader.	T$CMPC
Parallel interface to card punch.	O$AC03
Parallel interface to card punch and print on card.	O$AC15
Raw data mover.	T$PMPC

Magnetic Tape

Write EBCDIC to 9-track.	O$AM13
Read EBCDIC from 9-track.	I$AM13
Raw data mover.	T$MT

Phantom Processes

Switch logout notification on or off.	LON$CN
Read logout notification information.	LON$R
Start a phantom process.	PHNTM$

Process Suspension

Suspend a process for a specified interval.	SLEEP$
Suspend a process (interruptible).	SLEP$I

Query User

Prompt and read a name.	RNAM$A
Prompt and read a number (binary, decimal, octal, or hexadecimal).	RNUM$A
Ask question and obtain a YES or NO answer.	YSNO$A

Randomizing

Generate random number and update seed, based upon a 32-bit word size and using the Linear Congruential Method.	RAND$A
Initialize random number generator seed.	RNDI$A

Search Rules

Locate a file using a search list and open the file. Create a file if the file sought does not exist.	OPSR$
Locate a file using a search list and a list of suffixes. Open the located file, or create a file if the file sought does not exist.	OPSRS$
Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL.	SR$ABSDS

Add a rule to the beginning of a search list or before a specified rule.	SR$ADDB
Add a rule to the end of a search list or after a specified rule.	SR$ADDE
Create a search list.	SR$CREAT
Delete a search list.	SR$DEL
Disable an optional search rule. Used to disable rules that have been enabled using SR$ENABL.	SR$DSABL
Enable an optional search rule. Enabled rules can be disabled using SR$DSABL or SR$ABSDS.	SR$ENABL
Determine if a search rule exists.	SR$EXSTR
Free list structure space allocated by SR$LIST or SR$READ.	SR$FR_LS
Initialize all search lists to system defaults.	SR$INIT
Return the names of all defined search lists.	SR$LIST
Read the next rule from a search list.	SR$NEXTR
Read all of the rules in a search list.	SR$READ
Remove a search rule from a search list.	SR$REM
Set the locator pointer for a search rule.	SR$SETL
Set a search list using a user-defined search rules file.	SR$SSR

Semaphores

Release (close) a named semaphore.	SEM$CL
Drain a semaphore.	SEM$DR
Notify a semaphore.	SEM$NF
Open a set of named semaphores.	SEM$OP
Open a set of named semaphores.	SEM$OU
Periodically notify a semaphore.	SEM$TN
Return number of processes waiting on a semaphore.	SEM$TS

Wait on a specified named semaphore, with timeout.	SEM$TW
Wait on a semaphore.	SEM$WT

Sorting

Sort one file on ASCII key(s).	SUBSRT
Sort (multiple key types) or merge sorted files.	ASCS$$
Merge sorted files.	MRG1$S
Return next merged record to sort.	MRG2$S
Close merged input files.	MRG3$S
Sort one or several input files.	SRTF$S
Prepare sort table and buffers.	SETU$S
Get input records.	RLSE$S
Sort tables prepared by SETU$S.	CMBN$S
Get sorted records.	RTRN$S
Close all sort units.	CLNU$S
Heap sort.	HEAP
Partition exchange sort.	QUICK
Diminishing increment sort.	SHELL
Radix exchange sort.	RADXEX
Insertion sort.	INSERT
Bubble sort.	BUBBLE
Binary search or build binary table.	BNSRCH

Strings

Compare two strings for equality.	CSTR$A
Compare two substrings for equality.	CSUB$A
Fill a string with a character.	FILL$A
Fill a substring with a given character.	FSUB$A
Get a character from a packed string.	GCHR$A
Left justify, right justify, or center a string within a field.	JSTR$A
Locate one string within another.	LSTR$A
Locate one substring within another.	LSUB$A
Move a character between packed strings.	MCHR$A
Move one string to another.	MSTR$A
Move one substring to another.	MSUB$A
Compare two character strings.	NAMEQ$
Determine the operational length of a string.	NLEN$A
Rotate string left or right.	RSTR$A
Rotate substring left or right.	RSUB$A
Shift string left or right.	SSTR$A
Shift substring left or right.	SSUB$A
Test for pathname.	TREE$A
Determine string type.	TYPE$A
Return unique bit string.	UID$BT
Convert UID$BT output into character string.	UID$CH

System Administration

General System Administration

Change the user ID of the System Administrator.	CUS$CHANGE_ADMIN
Enable changes to the system attributes.	CUS$CHANGE_SYSTEM
Check System Administration Directory (SAD) hashing status for the system or a project.	CUS$CHECK_SAD
Close a SAD that has been opened.	CUS$CLOSE_SAD
Create a System Administration Directory (SAD).	CUS$CREATE_SAD
List the attributes of the overall system.	CUS$LIST_SYSTEM
Open an existing System Administration Directory (SAD).	CUS$OPEN_SAD
Rebuild the SAD for either the system or a project.	CUS$REBUILD_SAD
Check if the user is the System Administrator of the open SAD.	CUS$SA_MODE

Group Administration

Check if an ACL group is already a system ACL group or project ACL group.	CUS$CHECK_GROUP
Add an ACL group to the SAD.	CUS$GROUP
List the system and project ACL groups.	CUS$LIST_GROUP_NAMES
List the projects using an ACL group.	CUS$LIST_GROUPS_PROJECTS
List the users of a system or project ACL group.	CUS$LIST_GROUPS_USERS

Project Administration

Check if a project is on the system.	CUS$CHECK_PROJECT_ID
List the projects using an ACL group.	CUS$LIST_GROUPS_PROJECTS

List the attributes of a specific project.	CUS$LIST_PROJECT
Add, delete, or change a specific project.	CUS$PROJECT

User Administration

Check if a user is on the system or a member of a project.	CUS$CHECK_USER_ID
List the users of a system or project ACL group.	CUS$LIST_GROUPS_USERS
List the attributes of a specific user.	CUS$LIST_USER
List the users on the system or on a project.	CUS$LIST_USER_NAMES
Add, delete, or change a specific user.	CUS$USER
Check the network to see if a particular user ID is valid on other machines.	CUS$VERIFY_USER

System Information

General System Information

Return cold-start setting of the ABBREV switch.	ABSW
Determine if the routine is dynamically accessible.	CKDYN$
Return text of the specified system prompt.	CL$MSG
Return the model number of the Prime computer.	CPUID$
Return the current date and time.	DATE$
Return text representation of an error code.	ERTXT$
Return text representation of an error code for specified PRIMOS subsystem.	ER$TEXT
Return PRIMOS II information.	GINFO
Return the current PRIMOS system name.	GSNAM$
Return information on the system's list of logical disks.	LDISK$
Indicate if login-over-login is permitted.	LOV$SW

Return information about a PRIMOS line used for LAN terminal service.	NT$LTS
Return the operating system revision number.	PRI$RV
Determine access to a segment.	RSEGAC$
Check validity of a system name passed to it.	SNCHK$
Return the user number and count of users.	USER$

System Time Information

Return the CPU time since login.	CTIM$A
Return today's date, American style.	DATE$A
Return today's date as day of year (the Julian date).	DOFY$A
Return the disk time since login.	DTIM$A
Return today's date, European (military) style.	EDAT$A
Return the time of day.	TIME$A

System Status and Metering Information

Return data about a disk partition.	DS$AVL
Return data about a process's environment.	DS$ENV
Return data about file units.	DS$UNI
Return a variety of metering information.	G$METR

Timers

Set and read various timers.	LIMIT$
Create a timer.	TMR$CREA
Destroy a timer.	TMR$DEST

Set an absolute timer.	TMR$SABS
Set an interval timer.	TMR$SINT
Set a repetitive timer.	TMR$SREP
Cancel a timer.	TMR$CANL
Return the timer type and information.	TMR$GTMR
List the identifiers of the timers within a server.	TMR$LIST

User Information

Check that a process has a given amount of time slice left.	ASSUR$
Change login validation password.	CHG$PW
Expand a line using abbreviations preprocessor.	COM$AB
Generate a new login validation password.	GEN$PW
Validate a name.	IDCHK$
Determine whether a forced logout is in progress.	IN$LO
List the disks a given user is using.	LUDSK$
Log out a user.	LOGO$$
Return a list of devices that a user can access.	LUDEV$
Return the user's project identifier.	PRJID$
Return amount of CPU time used since login.	PTIME$
Validate syntax of a password.	PWCHK$
Display PRIMOS command prompt.	READY$
Return user number of initiating process.	SID$GT
Test whether current user is supervisor.	SUSR$
Display standard message showing times used.	TI$MSG
Return timing information and user identification.	TIMDAT
Return permanent time information.	TMR$GINF
Return current system time.	TMR$GTIM

Convert local time to Universal Time.	TMR$LOCALCONVERT
Convert Universal Time to local time.	TMR$UNIVCONVERT
List users with same name as caller.	UNO$GT
Return user type of current process.	UTYPE$
Validate a name against composite identification.	VALID$

User Terminal

Functions

Control functions for user terminal.	C$A01
Output ASCII to the user terminal or ASR punch.	O$AA01
Inhibit or enable CONTROL–P.	BREAK$
Get next character from terminal or command file.	C1IN
Get next character from command line until carriage return.	C1IN$
Move characters from terminal or command file to memory.	CNIN$
Read a line of text from the terminal or from a command file.	COMANL
Supervise the editing of input from a terminal or a command file (callable from C).	ECL$CC
Supervise the editing of input from a terminal or a command file.	ECL$CL
Read or set erase and kill characters.	ERKL$$
Output *count* characters to the user terminal followed by a line feed and carriage return.	TNOU
Output *count* characters to the user terminal.	TOVFD$
Read one character from the user terminal into Register A.	T1IB
Read one character from the user terminal.	T1IN

Write one character from Register A to the user terminal.	T1OB
Output *char* to the user terminal. The data type must be a 16-bit integer in F77.	T1OU
Input decimal number.	TIDEC
Input an octal number.	TIOCT
Input a hexadecimal number.	TIHEX
Output a six-character signed decimal number.	TODEC
Output a six-character unsigned octal number.	TOOCT
Output a four-character unsigned hexadecimal number.	TOHEX
Output carriage return and line feed.	TONL

Input From User Terminal

Read a character.	C1IN
Read a character.	C1IN$
Read a character, suppressing echo.	C1NE$
Read a line.	CL$GET
Read a specified number of characters.	CNIN$
Read a line into a PRIMOS buffer.	COMANL
Parse a command line.	RDTK$$
Read a character (function).	T1IB
Read a character (procedure).	T1IN
Read a decimal number.	TIDEC
Read a hexadecimal number.	TIHEX
Read an octal number.	TIOCT
Check for presence of characters in user's terminal output buffer.	TTY$OUT

Output to User Terminal

Print a standard error message from PRIMOS or a PRIMOS subsystem.	ER$PRINT
Print a standard error message.	ERRPR$
Provide free-format output.	IOA$
Provide free-format output, for error messages.	IOA$ER
Write characters to terminal, followed by NEWLINE.	TNOU
Write characters to terminal.	TNOUA
Write a signed decimal number.	TODEC
Write a hexadecimal number.	TOHEX
Write a NEWLINE.	TONL
Write an octal number.	TOOCT
Write a decimal number, without spaces.	TOVFD$
Write one character from Register A.	T1OB
Write one character.	T1OU

Control Output to User Terminal

Inhibit or enable BREAK function.	BREAK$
Return information about command output settings.	CO$GET
Switch input between the terminal and a file.	COMI$$
Switch output between the terminal and a file.	COMO$$
Control the way PRIMOS treats the user terminal.	DUPLX$
Read or set the erase and kill characters.	ERKL$$
Determine if there are pending quits.	QUIT$
Check for unread terminal input characters.	TTY$IN
Clear the terminal input and output buffers.	TTY$RS

Index of Subroutines by Name

UNITS$	Return caller's minimum and maximum file unit numbers.	II	4–124
UNO$GT	List users with same name as caller.	III	2–56
UPDATE	Update current directory (PRIMOS II only) (obsolete).	III	D–20
USER$	Return user number and count of users.	III	2–26
UTYPE$	Return user type of current process.	III	2–57
VALID$	Validate a name against composite identification.	III	2–59
WILD$	Return a logical value indicating whether a wildcard name was matched.	II	4–125
WRASC	Write ASCII.	IV	4–3
WRBIN	Write binary to any output device.	IV	4–7
WRECL	Write disk record.	IV	D–8
WTLIN$	Write a line of characters to a compressed ASCII file.	II	4–126
YSNO$A	Ask question and obtain a yes or no answer.	IV	11–7
Z$80	Clear double-precision exponent.	I	B–5

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Surveys

Reader Response Form
Subroutines Reference III: Operating System
DOC10082-2LA

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

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

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____
Position:_____
Company:_____
Address:_____
_____
_____Postal Code:_____

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