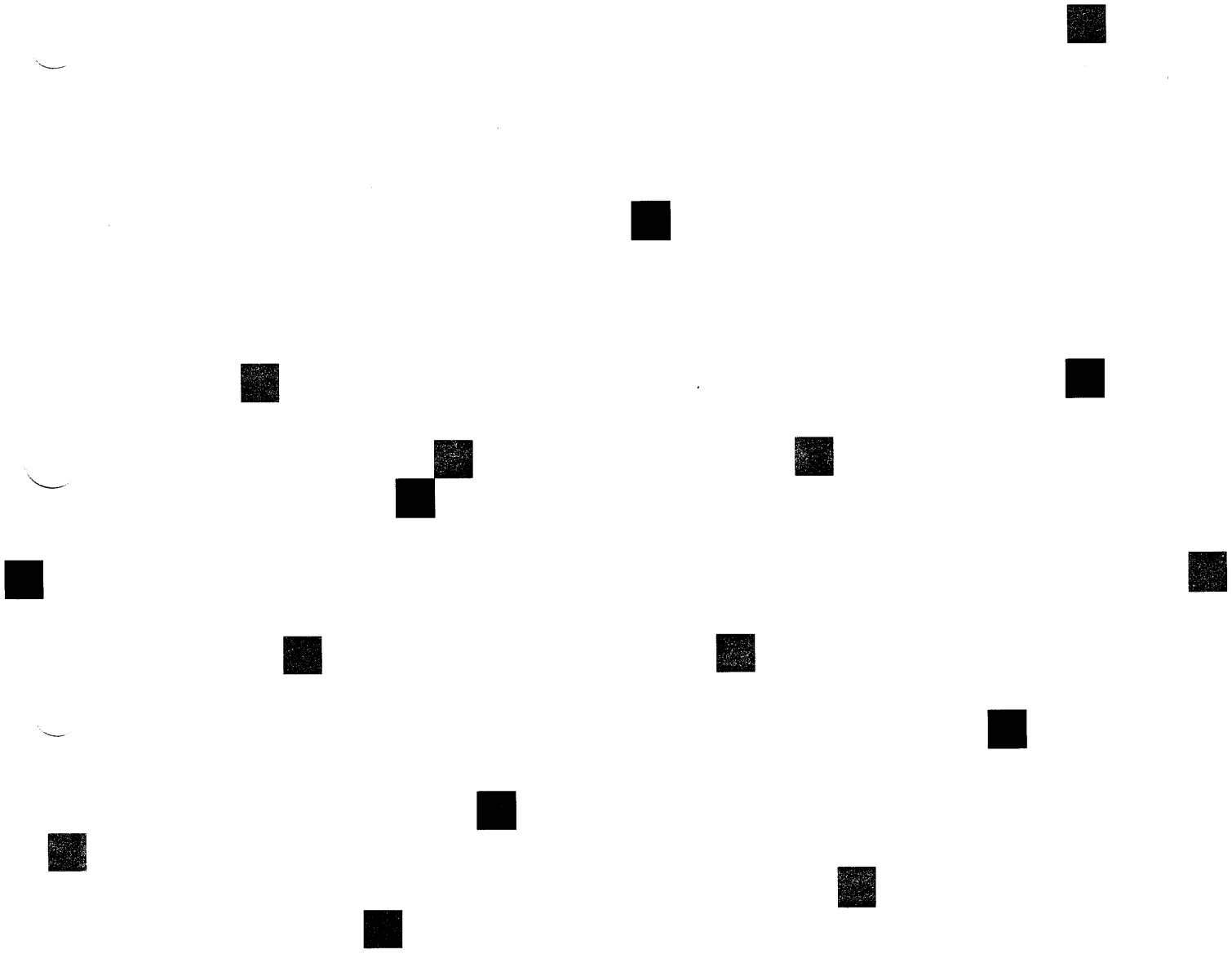


Prime®

Advanced Programmer's Guide Volume III Command Environment

Revision 19.4

DOC10057-1LA

Advanced Programmer's Guide Volume III: Command Environment

First Edition

by

James Craig Burley
and
Alice Landy

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.4.3 (Rev. 19.4.3).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

COPYRIGHT INFORMATION

CREDITS

Project Support	Len Bruns Margaret Taft
Editorial Support	Mary Callaghan
Graphic Support	Marjorie Clark Mike Moyle Bob Stuart
Production Support	Michelle Hoyt
Document Preparation	Nancy Cormier Mary Mixon

PRINTING HISTORY — Advanced Programmer's Guide,
Volume III: Command Environment

Edition	Date	Number	Software Release
Preliminary Edition	January 1985	DOC9229-1LA	19.4.0
First Edition	November 1985	DOC10057-1LA	19.4.3

In document numbers, L indicates loose-leaf.

CUSTOMER SUPPORT CENTER

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

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

HOW TO ORDER TECHNICAL DOCUMENTS

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

United States Only

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

International

Contact your local Prime subsidiary or distributor.

Contents

ABOUT THIS BOOK ix

Prime Documentation Conventions x
Calling Sequence Conventions xi

1 INTRODUCTION TO THE COMMAND ENVIRONMENT

Aspects of the Command Environment 1-2
Interactive Users 1-2
Command Input (COMINPUT) Files 1-3
Command Procedure Language (CPL) Programs 1-4
User-written Programs 1-6
User-written Functions 1-7
Applications 1-7
Types of Programs 1-8
Internal Commands 1-9
External Commands 1-10
The Command Interface 1-10
Limits on Program Invocation 1-13
Key Modules in the Command Environment 1-16
The Listener 1-17
The Command Prompter 1-19
The Command Line Reader 1-19
The Abbreviation Processor 1-19
The Command Processor 1-20
The Expression Evaluator 1-20
The Command Features Decoder 1-21
The Command Preprocessor 1-21
The Program Invokers 1-22
The Default On-unit 1-22

2 COMMAND LINE PROCESSING

Step 1: Handling the Command Separator Character 2-3
Step 2: Evaluation of Function and CPL Variable References 2-4
Step 3: Removal of Null Tokens 2-4
Step 4: Determination of Command Name 2-4
Step 5: Determination of Command Type 2-5

Step 6: Determination of Command Iteration Features 2-5
Step 7: Expansion of Simple Iteration 2-5
Step 8: Expansion of Treewalking 2-6
Step 9: Expansion of Wildcard Specifications 2-6
Step 10: Expansion of Name Generation Patterns 2-7
Invocation 2-8

3 PROGRAM EPF CALLING SEQUENCE

Types of Calling Sequences 3-2
Program Calling Sequence 3-3
Command Calling Sequence 3-3
Command Function Calling Sequence 3-6
The ALS$RA Subroutine 3-9
The ALC$RA Subroutine 3-10
Detailed Command Calling Sequence 3-15
Command Processing Information 3-19
Complete Calling Sequence 3-26

4 INVOKING PROGRAMS FROM WITHIN PROGRAMS

Commands, Programs, and Functions 4-2
Deciding Which Interface to Use 4-6
The CP$ Subroutine 4-9
Using CP$ to Invoke a Command or Program 4-9
Using CP$ to Invoke a Function 4-13
Error Codes From CP$ 4-17
The EPF$RUN Subroutine 4-18
Error Codes From EPF$RUN 4-26
The EPF$INVK Subroutine 4-27
Error Codes From EPF$ Subroutines 4-40
The FRE$RA Subroutine 4-45
Sample Programs 4-47
If a Program Invokes Itself 4-54
Terminal Input and Output 4-55

5 THE COMMAND PROCESSOR STACK

What the Command Processor Stack Is Used For 5-1
Command Levels 5-2
The Listener 5-2
The RDY Command 5-4
The RELEASE_LEVEL Command 5-5
The DUMP_STACK Command 5-11
The INITIALIZE_COMMAND_ENVIRONMENT Command 5-12
The REENTER Command 5-13
Mini-Command Level 5-15
What Control-P Actually Does 5-16
If Your Program Catches QUIT$ 5-16

6 THE RECURSIVE COMMAND ENVIRONMENT

What Is a Recursive Resource? 6-1
What Is a Dynamic Resource? 6-2
What Is a Static Resource? 6-2
The Cache Attach Point as a Static Resource 6-2
Other Static Resources 6-3

INDEX X-1

About This Book

The Advanced Programmers's Guide is intended for programmers who are experienced with Prime 50 Series ™ systems, have read the Prime User's Guide (DOC4130-4LA) and Programmer's Guide to BIND and EPFs (DOC8691-1LA), are familiar with the Subroutines Reference Guide (DOC3621-190) and its first update package (UPD3621-31A), are experienced in at least one high-level language supplied by Prime (preferrably PL1G or FTN), and who have an understanding of the architecture of Prime systems as described in the Prime 50 Series Technical Summary (DOC6904-191) and in the System Architecture Guide.

This guide is divided into several volumes.

- Volume 0 of this guide describes new features of interest to readers of this guide. It also describes standard error codes used by PRIMOS™, along with their messages and meanings.
- Volume I describes Executable Program Formats (EPFs).
- Volume II describes the PRIMOS File System.
- Volume III (this volume) describes the PRIMOS Command Environment.

Designed for systems-level programmers, this guide describes the lowest-level interfaces supported by PRIMOS and its utilities. Higher-level interfaces not described in this guide include:

- Language-directed I/O
- The applications library (APPLIB)

- The sort packages (VSRTLI and MSORTS)
- Data management packages (such as MPLUSLB and PRISAMLIB)
- Other subroutine packages

All the above interfaces are described in other manuals, such as language reference manuals and the Subroutines Reference Guide.

This guide documents the low-level interfaces for use by programmers and engineers who are designing new products, such as language compilers, data management software, electronic mail subsystems, utility packages, and so on. Such products are themselves higher-level interfaces, typically used by other products rather than by end users, and therefore must use some or all of the low-level interfaces described in this guide for best results.

Because of the technical content of the subjects presented in this guide, it is expected that this guide will be regularly used only by project leaders, design engineers, and technical supervisors rather than by all programmers on a project. Most of the information in this guide deals with interfaces to PRIMOS that are typically used only in small portions of a product, and with overall product design issues that should be considered before coding begins. Once the product is designed and the PRIMOS interfaces are designed and coded, a typical product can then be written by programmers whose knowledge of these issues is minimal. Of course, this statement is predicated on the assumption that widely accepted programming practices, such as modular, or structured, programming, functional and design specifications, and thorough unit debugging and testing, are employed.

PRIME DOCUMENTATION CONVENTIONS

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

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. These can be entered in either uppercase or lowercase letters.	SLIST

lowercase	In command formats, words in lowercase letters indicate items for which the user must substitute a suitable value.	LOGIN user-id
abbreviations	If a command or statement has an abbreviation, it is indicated by underlining. In cases where the command or directive itself contains an underscore, the abbreviation is shown below the full name, and the name and abbreviation are placed within braces.	LOGOUT { SET_QUOTA { SQ }
underlining in examples	In examples, user input is underlined but system prompts and output are not.	OK, RESUME MY_PROG This is the output of MY_PROG.CPL OK,
Brackets [ ]	Brackets enclose one or more optional items. Choose none, one, or more of these items.	SPOOL [-LIST [-CANCEL]
Braces { }	Braces enclose a list of items. Choose one and only one of these items.	CLOSE { filename { ALL }
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	DIM array (row,col)
Hyphen -	Wherever a hyphen appears as the first letter of an option, it is a required part of that option.	SPOOL -LIST

CALLING SEQUENCE CONVENTIONS

This guide provides many illustrations of calling sequences as a handy reference for readers while they read the guide. These illustrations are not intended to replace the Subroutines Reference Guide for complete reference information on subroutines. For example, this guide may not show all of the various forms of invoking a subroutine if only a few forms apply to the topic being discussed.

Calling sequences of subroutines, programs, and functions are illustrated in this guide in summary form. Each calling sequence occupies one full page. The subroutine, or procedure, name is listed in the middle of the page, followed by dummy parameter names, separated by commas, listed in parentheses. This is the basic calling sequence for the procedure.

Above the calling sequence are the input arguments; below the calling sequence are the output arguments. Arrows are drawn to or from the dummy parameter names to indicate the flow of information and also to visually connect parameter names to the information on the parameters. Each input or output parameter includes the following information:

- A description of the argument
- The datatype of the argument

For the description of the argument, a short description may be given; in some cases, such as for keys, a value or a list of choices of values is given; occasionally, an illustration of the format of the input or output argument is provided. The choice is designed to prove the most useful when the reader uses the subroutine in one or more programs somewhat frequently.

For the datatype of the argument, a datatype description language is defined specifically for this guide. Readers must convert the datatype description language used here to the appropriate language. This guide often includes PL/I-G or FORTRAN versions of structures in addition to the datatype description language.

In addition to the arguments, or parameters, for the procedure, procedures that are functions return a function value. In this case, the value and its datatype is illustrated below the name of the procedure itself. The meaning and datatype of the function value is essentially the same as for parameters.

There are two main purposes to the format used in this guide to illustrate calling sequences:

- To illustrate the calling sequence for a single type of function performed by a procedure
- To show the relationships between interdependent parameters in a calling sequence

The first purpose is addressed by separating information on a multipurpose subroutine such as PRWF$$ into several different calling sequence descriptions, one for reading a file, another for writing a file, another for positioning a file, and so on.

The second purpose is addressed by providing dotted lines between related arguments in calling sequence illustrations. Most often, such relationships involve a character string parameter whose length is specified by another parameter in the calling sequence. Another example is the size of an array parameter that is specified by another parameter.

Datatypes

The following datatypes, and their PL/I-G and FORTRAN equivalents, are used throughout this guide:

Datatype	PL/I-G	FORTRAN
HALF INT	FIXED BIN(15)	INTEGER*2
FULL INT	FIXED BIN(31)	INTEGER*4
n STRING	CHARACTER(n)	INTEGER*2 ((n+1)/2)
<=n STRING	CHARACTER(n) VARYING	INTEGER*2 ((n+3)/2)
2 ((n+3)/2)		
n BIT	BIT(n)	INTEGER*2 ((n+15)/16) w/masking
PTR	POINTER and ADDR()	INTEGER*2 (3) and LOC()
STRUC	(see Note 1)	(see Note 1)
ARRAY(n)	(see Note 2)	(see Note 2)

Notes

1. Structures are usually illustrated in the same figure or in another figure, or their declarations are provided in a page near the figure. They are also known as "record" data types in other languages.

2. Arrays are either a constant length (n is indicated in parentheses) or a varying length set controlled by another parameter or by a subfield in another parameter. Varying length arrays have dotted lines from the word ARRAY to the parameter (or its subfield) that controls the length of the array.

If you are unsure as to the meaning of a keyword, arrow, or other illustrative mark, consult the Subroutines Reference Guide for more precise and complete information on the subroutine or data structure.

Keys

Some Prime-supplied interfaces take a key argument as an input-only argument. Your program sets key to specify the precise operation to be performed by the interface. In most figures that involve a key argument, only a list of valid (or appropriate) values for the argument is provided in the form of the keyword names for the keys; these keyword names, once learned, are easy to associate with the corresponding function. For example, the k$read key specifies a read operation.

When the construction of a key is complex, two or more lists of keywords are often shown, enclosed in braces, with + signs to indicate addition. As with command formats, the braces indicate that you should pick one keyword from each list in braces; the + signs indicate that you should add, in the program, the resulting keywords. For example, your program might specify a key value of k$rdwr+k$ndam+k$getu.

To define key definition keywords for your program, which have names beginning with K$, use a %INCLUDE or $INSERT statement to insert the appropriate SYSCOM>KEYS.INS.language file into your program. See the Subroutines Reference Guide for more information on this topic.

Standard Error Code

Most interfaces include a standard error code as a parameter. This is a HALF INT value returned by the interface to indicate the degree of success encountered by the interface. When provided by an interface, your program should always check this value to ensure that it is 0 (zero) after each call to the interface — a value of 0 means a successful call. Other values can mean either an error or just a condition worth noting. Volume 0 contains a list of all standard error codes along with descriptions of their various meanings within PRIMOS.

To define standard error code keywords for your program, which have names beginning with E$, use a %INCLUDE or $INSERT statement to insert the appropriate SYSCOM>ERRD.INS.language file into your program. See the Subroutines Reference Guide for more information on this topic.

Side Effects

Where appropriate, the side effects of an interface are listed in the lower left-hand corner of the corresponding figure.

General Coding Guidelines

When writing programs that use any Prime-supplied subroutines, observe the following guidelines to ensure that your programs continue functioning normally on subsequent revisions of PRIMOS:

- Reserved or undefined information returned to your program by a Prime-supplied interface subroutine must be ignored. For example, if a 16-bit halfword contains one defined bit and fifteen reserved bits, your program must mask off the fifteen reserved bits before analyzing the halfword to determine the value of the one defined bit.

- Reserved or undefined information passed by your program to a Prime-supplied interface must contain all zeroes, except where otherwise specified.

1

Introduction to the Command Environment

The PRIMOS command environment is a collection of subroutines and interfaces that provide a single, flexible, and efficient command interface for:

- Interactive users, who issue commands and then usually wait for a response before issuing subsequent commands
- Command input (COMINPUT) files, which contain simple command scripts
- Command Procedure Language (CPL) programs, which contain a mix of commands and CPL directives; these interpreted programs can issue commands and make decisions based on the results of those commands
- User-written programs, which may or may not wish to take advantage of certain features of the command environment such as wildcarding, iteration, treewalking, and name generation
- User-written functions, which can be used just like CPL command functions in that they can return a string value to the caller
- Applications, which may invoke other commands, programs, and functions that are external to the application

Because the command environment allows the mixing and matching of its many features, you can build very powerful packages by combining command environment features. For example, you may be able to build a powerful application using small, easy-to-manage programs and functions, instead of having to construct a single, large, monolithic program.

This chapter introduces you to the features of the command environment of particular interest to advanced programmers. It begins by examining how the various aspects of the command environment serve the needs of the six types of "users" listed above. It then discusses the three types of program recognized by the command environment and the way in which the command environment interfaces to each. It discusses per-site limits that can be imposed on program invocation. Finally, it explains the structure of the command environment by listing and discussing the key modules of the command environment.

ASPECTS OF THE COMMAND ENVIRONMENT

This section describes the types of users served by the command environment and the features of most interest to each. The command environment features, themselves, are explained briefly in this section; they are discussed in more detail later in this chapter.

Interactive Users

Interactive users are those users who enter commands at their terminals; usually, they then wait for a response before issuing subsequent commands. Their needs are:

- To know, as soon as possible, whether the command they issued has succeeded
- To be able to issue powerful commands using the fewest possible keystrokes
- To perform certain operations on multiple targets (such as a group of files), without having to enter the same command repeatedly

For these users, the command environment provides:

- An error-reporting facility with which programs can display identical, and therefore familiar, error messages for identical error conditions
- A command prompt that immediately lets the user know whether the previous command succeeded (OK,) or failed (ER!)

- An abbreviation facility, whereby a user can specify that a particular command or keyword is a substitute for a more lengthy and perhaps more complicated command, keyword, or sequence of commands and keywords

- A sophisticated command preprocessor that provides treewalking, wildcarding, and iteration facilities; these allow a user to easily specify sets of file system objects (either listed by the user or related by name or by common parent directory) without having to retype the command and its options for each specified object

Interactive users who repeatedly issue a simple set of commands may use command input files; such files are described in the next section.

Command Input (COMINPUT) Files

Command input files contain simple command scripts that, when invoked by an interactive user via the COMINPUT command, substitute for the user's interactive input of commands. Unlike an interactive user, a command input file does not have the ability to "look" at the results of a particular command and decide how to proceed next. The needs of a command input file are:

- To be able to record output of the command input session for later perusal by the user

- To be automatically stopped after a fatal error, so that subsequent commands do not cause further problems

To support the needs of command input files, the command environment provides:

- A command output facility (the COMOUTPUT command), allowing command input and output to be written to a file as well as (or instead of) to the user terminal

- An error-reporting facility that allows commands and programs to indicate whether they completed successfully

- An error-detection facility that automatically suspends command input upon detection of a fatal error

Although command input files are useful for simple command sequences that change little or not at all between invocations, they do not provide more sophisticated features, such as allowing a user to provide one or more arguments to the command input file that modify its actions or allowing the command file itself to choose a course of action depending on the results of invoking a command or program. Prime's Command Procedure Language (CPL), described next, provides these features.

Command Procedure Language (CPL) Programs

A Command Procedure Language (CPL) program contains a combination of commands and CPL directives. Commands, such as COPY, F77, BIND, and RESUME, typically perform the actual work of a CPL program. CPL directives, such as &ARGS, &IF, &SET_VAR, and &RETURN, control the execution of commands within the program. CPL programs may use CPL variables and functions throughout; both are similar to programming language constructs in that they substitute actual values for themselves at program runtime. CPL variables substitute values assigned at program runtime, while function references invoke PRIMOS-resident functions or call other programs at program runtime to determine what values to substitute.

CPL programs are sometimes substituted for existing compiled programs; for example, a CPL program may provide a program-use log or a more user-friendly interface.

The needs of a CPL program are:

- To be invoked just as if it were a compiled program
- To be able to record output of the CPL program session for later perusal by the user or for later interpretation by the CPL program
- To be able to execute PRIMOS commands without repetition of the PRIMOS OK, prompt, thus avoiding filling the user's screen with OK, prompts
- To be able to intercept and analyze errors encountered by commands and programs invoked by the CPL program, in order to determine the next course of action
- To be able to report errors encountered by the CPL program to the invoker of the program, in a form useful to both the interactive user and to another program, either of which may invoke the CPL program
- To be able to invoke other CPL programs
- To be able to return a value as a result of the CPL program when the program is to be used as a function

To support the needs of CPL programs, the command environment provides:

- A program invocation interface (the RESUME command) that invokes both compiled programs and CPL programs, depending upon which it finds, so that the user is not necessarily aware of the type of program being invoked

- A command output facility (the COMOUTPUT command), allowing command input and output to be written to a file as well as (or instead of) being written to the user terminal

- A command line reader that does not display PRIMOS prompts when it is reading commands from a CPL program

- An error detection and interception directive (the &SEVERITY directive) that allows a CPL program to intercept fatal command or program errors without necessarily resulting in the abnormal termination of the CPL program, and that allows the CPL program to analyze the error code (the %SEVERITY$% variable) returned by a command or program

- An error condition interception directive (the &ON directive) that allows a CPL program to intercept program runtime errors (such as ACCESS_VIOLATION$) in addition to program interrupts (such as QUIT$, and LOGOUT$) and that allows the CPL program to clear the condition or to continue the signaling of the condition

- A directive (the &RETURN directive) that allows a CPL program to return a severity code to the calling program, indicating whether an error condition was encountered, and to specify an error message, indicating the nature of the error, to be displayed on the user's terminal.

- The ability to invoke other CPL programs via the RESUME command exactly as if they were compiled programs

- A directive (the &RESULT directive) that allows a CPL program to return a text string as the result of the CPL program, for use when the CPL program is designed as a function

See the CPL User's Guide for complete information on how to write CPL programs.

Because CPL provides many constructs found in structured programming languages such as PL/I, it is useful for rapid development of utilities and programs. In particular, CPL can be an appropriate language for the development of a prototype utility. CPL programs tend to be easy to understand and maintain because they are interpreted rather than compiled and because the debugging of such programs is typically straightforward (and is assisted by other CPL directives, such as &DEBUG and &WATCH).

However, most CPL programs run faster when converted to one of Prime's compiled languages, such as PL/I-G. To ease the conversion of CPL programs to other Prime-supplied languages, the command environment provides compiled programs with the same abilities as CPL programs, as described next.

User-written Programs

User-written programs are compiled programs that are linked using Prime's BIND, LOAD, or SEG linker and executed using the RESUME or SEG command. These programs may or may not wish to take advantage of certain features of the command environment such as wildcarding, iteration, treewalking, and name generation. When it is called upon to execute a program, the command environment detects which of its features the program wishes to use. The needs of a user-written program that interfaces to the command environment are:

- To be able to execute PRIMOS commands
- To be able to intercept and analyze errors encountered by commands and programs invoked by the program to determine the next course of action
- To be able to report errors encountered by the program to the invoker of the program, in a form useful to both the interactive user and to another program, either of which may invoke the program
- To be able to invoke other programs
- To be able to determine what command processing features, such as wildcarding and iteration, are being used to invoke the program

The command environment provides facilities to programs built as EPFs that address all of the above needs. For static-mode programs, which are linked via SEG or LOAD, the command environment provides a limited set of facilities. Facilities provided by the command environment for use by compiled programs are:

- An interface (the CP$ subroutine) to the command processor that allows a running program to invoke a PRIMOS command, a CPL program, or another compiled program
- A returned severity code from the CP$ subroutine interface that represents the level of success encountered by the invoked command or program
- A program interface that allows a program to indicate its level of success by modifying a severity code variable, which was passed to it when the program was invoked by the command processor
- An interface (the ERRPR$ and ERTXT$ subroutines) that allows a program to display an error message corresponding to a standard PRIMOS error code, providing the user with consistent error messages for similar errors
- An interface (the EPF$RUN subroutine) that allows a program to invoke another program EPF

- A program interface that allows a program to determine, by analyzing a structure passed to it by the command environment, which command preprocessing features (such as wildcarding and iteration) are involved in the invocation of the program

While compiled programs are generally built to perform some task and not to return the results of a calculation, a program may be designed to return a result to the invoker of the program. These programs, called functions, are described next.

User-written Functions

Like CPL command functions, user-written functions return a string value to the caller. A user-written function must be a program EPF; it cannot be a static-mode program. (Any program EPF that returns a string value to its caller is a function.) To allow the writing of such functions, the command environment:

- Provides an interface (the ALC$RA and ALS$RA subroutines) that allows a program to allocate memory that is to contain the returned text string
- Passes to a program a pointer that the program sets to point to the returned text string allocated by ALC$RA or ALS$RA
- Provides an interface (the FRE$RA subroutine) that allows a program that calls a function, whether a CPL program or a program EPF, to deallocate the memory associated with a returned text string after using the string

Some programs may not fit into either the category of a program that performs a task or the category of a program that is a function. Such programs may invoke other programs, provide specially tailored user interfaces, manage data bases, and so on. These applications are described next.

Applications

This guide uses the term application to describe a program that may invoke other commands, programs, and functions that are external to the application. In addition, an application may provide a user with a specially tailored interface, such as an interactive menu-driven file management system. The needs of an application are similar to the needs of a user-written program, with the additional requirement that it be able to invoke a function and be able to make use of the returned value of that function.

The command environment allows a program to invoke a function by returning to the calling program a pointer to a structure containing the returned text string. In addition, the command environment provides an interface (the FRE$RA subroutine) that allows a program to deallocate the structure containing the returned text string once it has been used.

Applications may have additional requirements:

- To be able to repeatedly invoke a particular EPF without repeatedly mapping and unmapping the EPF
- To be able to modify the procedure code of an EPF once it is in memory, such as when an application allows interactive debugging of an EPF
- To be able to retrieve information on an EPF, such as how many procedures and segments it needs, its version number, the version of BIND used to link the EPF, and so on

The command environment satisfies these requirements by providing an EPF interface that consists of several subroutines which, called separately, allow an application to exercise more control over how and when an EPF is passed through its phases before being executed. (See Volume I of this series for a description of the phases in the life of an EPF.) Included in this interface is a subroutine, EPF$CPF, that allows a program to retrieve information on an EPF similar to that displayed by the LIST_EPF command.

TYPES OF PROGRAMS

Several interfaces exist between the command environment and the programs it handles. These interfaces differ in their complexity and in the capabilities each provides; each is designed to handle a particular type of program.

To the command environment, a program is either a simple program, a command, or a function.

A simple program is a program that does not take command line arguments (such as filenames, options, and so on). Moreover, a simple program does not indicate whether it succeeded or failed; it is always presumed to succeed.

A command is a program that accepts command line arguments or that indicates whether it succeeded or failed by returning a severity code, calling SETRC$, or calling ERRPR$. Most commands both accept command lines and return severity codes.

A function is a command or program that returns a character string that serves as the value of the function invocation. Most functions are internal to PRIMOS and are used by CPL programs, and are therefore called CPL command functions.

Some of these internal functions are usable as commands. For example, the DATE command function can be invoked as a function or as a program:

```
OK, TYPE 'It is '[DATE -DOW]', '[DATE -CAL]' at '[DATE -AMPM]'.'
It is Tuesday, March 5, 1985 at 8:06 AM.
OK, DATE
05 Mar 85 08:06:20 Tuesday
OK,
```

Internal CPL command functions, like internal PRIMOS commands, are not stored on the disk as programs, but are part of PRIMOS itself.

Another distinction made by PRIMOS is whether a command is internal or external. An internal command resides in PRIMOS itself rather than on disk; therefore, it is always available for use by users and by programs. An external command resides on disk, either in the top-level directory named CMDNC0 (a historical name that stands for CoMmanDs, Non-Chargeable, number 0) or elsewhere on disk. Because an external command is a file on disk, you can:

- Delete the file, which makes it unavailable to users and programs
- Change the name of the file, which also changes the name of the command
- Create a new file, which creates a new command
- Set access on the file, restricting its use to certain users, which also restricts use of the command to those users

A program, command, or function can be either internal or external, but not both. A few special internal commands are used to execute external commands, and these internal commands are therefore treated specially by the command environment.

Internal Commands

Internal commands, such as ATTACH and RESUME, are recognized by the command processor as being internal. They cause certain subroutines internal to PRIMOS to be invoked. CPL command functions, such as DATE and ATTRIB, are also considered to be internal.

External Commands

When you issue a command that is neither an internal PRIMOS command nor an internal CPL command function, PRIMOS looks in the top-level directory named CMDNCO for a program with the same name as the command. Programs in CMDNCO are called external commands. They are CPL, EPF, or static-mode programs that have been placed in CMDNCO by Prime or by your System Administrator.

Because external commands and programs differ only in where they are stored, issuing the ED command is effectively the same as typing:

```
RESUME CMDNCO>ED
```

PRIMOS determines what type of program you are invoking by appending various file suffixes to the program name and checking to see if a file with the resulting name exists. The file suffix is then used to determine what type of program is being invoked. The file names are searched in the following order:

File Name	Program Type
command-name.RUN	EPF
command-name.SAVE	static-mode
command-name.CPL	CPL
command-name	static-mode

Note

If you supply the .RUN, .CPL, or .SAVE suffix when you invoke a program, PRIMOS searches for only that particular program. For example, typing RESUME MYPROG.CPL only causes MYPROG.CPL to be invoked as a CPL program; PRIMOS does not search for MYPROG.RUN or MYPROG.CPL.RUN in this case.

THE COMMAND INTERFACE

Because PRIMOS includes the command processor, the interface between the command processor and commands (programs) is defined by PRIMOS. This interface is described in detail in Chapter 3. In summary, the interface has five levels of complexity:

1. Program invocation. The program being invoked takes no arguments and returns no value; hence, it is a program, rather than a function, and it ignores any command line passed to it. No severity code is returned, so a severity code of 0 (successful completion) is assumed.

2. Command invocation. The program being invoked accepts a command line as an argument, and returns only a severity code; hence, it is a command program, rather than a function.

3. Function invocation. The program being invoked accepts:

   - A command line
   - An indication of whether the program being invoked is expected to return a value, used when the program can run as a program or a function

   Like commands, functions return severity codes.

   When the program is invoked as a function, it also returns the result of the function as a character string. It does this by allocating a structure into which it places the returned value and then returning a pointer to that structure.

4. Detailed command invocation. The program accepts the same information accepted by a command plus a description of the command state (including the command name, information on whether wildcards, treewalking, and other command preprocessing features have been selected, and so on). As with a command, a detailed command returns a severity code.

5. Complete command invocation. The program accepts all of the information accepted by both a function and a detailed command.

For most programs that invoke other programs, there are really only two forms of invocation: command invocation and function invocation. In general, programmers consider all five levels of complexity listed above only when designing program interfaces. For this design task, the availability of several levels of complexity combines the greatest amount of power with the ability to use simple interfaces when the power is not needed.

If one program is being written to invoke another, the invoking program does not need to concern itself with the level of complexity in the invoked program's interface. For example, suppose program A is to invoke program B. Assume that B is a function, and A invokes it as a command (that is, A does not ask for a returned text string). In this case, A is asking B to do less than B is capable of doing. B recognizes this and does not return a function value. On the other hand, suppose that B is a command and that A invokes it as a function. In this case, A asks B to do more than B is capable of doing. B will not recognize this, while A will interpret the lack of result as a successful invocation without any returned function value.

There are cases where the level of complexity is high; these generally involve sophisticated combinations of function invocation, CPL local variable pointers, and, sometimes, command line iteration.

The Command Line

An EPF accepts a command line as a character string in the calling sequence of its main entrypoint. How the EPF interprets the command line is entirely up to the programmer of the EPF. However, using the standard PRIMOS command line processing subroutines such as CL$PIX is recommended so that the EPF can take advantage of command line features such as wildcarding, treewalking, and so on, without any special programming.

The Severity Code

An EPF returns a severity code as a number in the calling sequence of its main entrypoint. When a program calls another program, the calling program interprets the severity code returned by the called program however it chooses. No special action upon receiving a positive severity code is required by PRIMOS, although the calling program typically takes corrective action or logs the error.

The Returned Character String

An EPF returns a character string by allocating memory for it, writing it into the allocated memory, and returning a pointer to the memory in the calling sequence of its main entrypoint. An EPF returns a value only if the invoking program has requested it by setting to '1'b a flag in the calling sequence of the main entrypoint of the EPF.

The returned character string can be used by the caller. For example, a program named USER_ID might return the username of the invoking user. USER_ID might be used in a CPL program as follows:

```
TYPE Your username is [RESUME PROGRAMS>USER_ID]
```

Programs that return such text strings are called functions. They are EPFs or CPL programs; a static-mode program cannot return a text string.

The Command Processing Information

An EPF obtains information on the command processing performed to invoke it by accepting a structure as an argument in the calling sequence of its main entrypoint. This structure, which is built by the command processor or by the invoking program, communicates the following information:

- The command used to invoke the program EPF (the name of the program)
- A pointer to CPL variables local to the CPL program that invoked the program EPF or one of its ancestors
- Information on the iteration features and options enabled during the invocation of the program EPF, such as wildcarding, object type selection, treewalking, and so on

Most program EPFs that use the information in this structure are probably going to use only the command name or the pointer to CPL local variables. Only programs that determine their behavior according to the manner in which they are invoked make use of information on iteration features.

LIMITS ON PROGRAM INVOCATION

Each system enforces the following resource limits on the invocation of programs from within programs or from command level:

- The limit on the maximum number of programs at a given command level (program breadth)
- The limit of the maximum number of dynamic segments
- Resource limits on memory utilization

Each program EPF takes up at least one additional dynamic segment; typically, each takes up two segments.

Your System Administrator sets limits on the number of programs you can invoke from within other programs at a given command level, and also on the number of dynamic segments you can use. The LIST_LIMITS command displays these limits, in addition to limits on the number of command levels and the number of static segments.

For example:

```
OK, LIST_LIMITS

Maximum number of command levels: 10
Maximum number of program invocations: 20
Maximum number of private static segments: 40
Maximum number of private dynamic segments: 50

OK,
```

You may also encounter situations in which, even though you are not exceeding limits placed on your process by the System Administrator, system-wide resources (such as segments) are exhausted. The remainder of this section describes what happens when per-user limits are reached and when system-wide resources are exhausted.

When Per-user Limits Are Reached

An attempt to exceed the limit on the maximum number of program invocations causes CP$ or EPF$RUN to return an error code of E$ECEB (Exceeding command environment breadth).

Even if the program is successfully invoked, the limit on dynamic segments may be reached before the program acquires sufficient memory to complete successfully; or, system resources may be exhausted before this point. If the limit on dynamic segments, as set by your System Administrator, is exceeded, one of several messages may be displayed.

Use the LIST_SEGMENTS command in conjunction with the LIST_LIMITS command to determine how many dynamic segments you are using and how many you can use.

For example, if you exceed the limit while PRIMOS is trying to resolve a dynamic link to a library EPF that it is unable to map and initialize due to insufficient memory, the following message is displayed:

```
Error: condition "LINKAGE_ERROR$" raised at 4000(3)/2101.
"Not enough segments." while attempting
dynamic link to entrypoint "SUBR" .
```

If the problem occurs while executing the external login program, the message appears as in the following example:

```
Condition "LINKAGE_ERROR$" raised at 4000(3)/3354 while in External
Login.  Please report this message to your system administrator.
```

If the system is unable to allocate process-class storage due to a lack of sufficient dynamic segments, the following message is displayed:

```
No space available from process class storage heap.
```

The condition SYSTEM_STORAGE$ is then raised, which may cause another message to be displayed by the default on-unit (or the on-unit for errors during external login).

A message you might see if insufficient space is available for memory allocation is:

```
STORAGE raised at 41(3)/112533
(insufficient space for ALLOCATE)

ERROR raised at 41(3)/112533
(no on-unit for STORAGE)
```

If the default on-unit can identify the program attempting the allocation, the message appears as follows:

```
STORAGE raised in PATH$ at 4355(3)/50541
(insufficient space for ALLOCATE)

ERROR raised in PATH$ at 4355(3)/50541
(no on-unit for STORAGE)
ER!
```

If there is insufficient memory to map a program EPF to memory for execution (as a result of a command, for example), the following message is displayed:

```
Not enough segments. program-EPF-name (std$cp)
ER!
```

In some cases, the error occurs at a point in which PRIMOS cannot recover without reinitializing your command environment, in which case the following message informs you of this event:

```
User environment re-initialized. (FATAL$)
```

Exceeding limits as set by your System Administrator usually means that, to successfully run your program, you must persuade your System Administrator to increase your limits. If this is not possible, you must reduce the number of dynamic segments or program invocations used by your program.

When System-wide Resources Are Exhausted

Even if you do not reach your limits on program invocation or dynamic segments, you may encounter a system-wide resource restriction. For example, if the system runs out of segments, the following message is displayed:

```
Error: condition "NO_AVAIL_SEGS$" raised at 4464(3)/104425.
```

Another possibility is that the system could run out of disk storage for virtual memory. For example:

```
Error: condition "PAGING_DEVICE_FULL$" raised at 4464(3)/104425.
```

In both cases, you should issue the ICE command and try running your program again. ICE resets your stack history and returns all of your segments to the system-wide free segment pool. When you run your program again at this point, you are using only those segments needed by your program. If the condition recurs, then the system is unable to run your program. This is normally a temporary condition.

If it seems that you cannot run your program without encountering such resource restrictions, contact your System Administrator about adding segments and/or paging space to the system.

KEY MODULES IN THE COMMAND ENVIRONMENT

The command environment is made up of many subroutines and data structures in PRIMOS. Some of these are explained in this section, because an understanding of them may help you learn about making sophisticated use of the command processor.

The key modules in the command environment are:

- The listener, which inputs commands from the user and executes them
- The command prompter, which displays a prompt at the user's terminal so that the user knows the system is awaiting input

- The command line reader, which reads commands from the user
- The abbreviation processor, which expands short character sequences in a command line interactively input by the user and replaces them with longer, more complex sequences
- The command processor, which executes commands
- The expression evaluator, which resolves, in a command line, references to functions and to CPL variables
- The command features decoder, which determines whether each command feature is enabled or inhibited for a particular command
- The command preprocessor, which performs all forms of iteration (such as simple iteration, treewalking, wildcards) and name generation
- The program invokers, each of which invokes a particular type of program (internal, EPF, static-mode, or CPL)
- The default on-unit, which is invoked for signaled conditions that are not caught by the running program, and which is often responsible for a new invocation of the listener

The actions of each of these key modules are summarized below. Not all of these modules are necessarily invoked for each command line — the command processor is able to detect whether a particular module, or set of modules, can be skipped because the command line does not require the features provided by that module. For example, suppose a command line contains no wildcard characters (@, +, or ^), no name generation character (=), no hyphens (-) to indicate options, such as object type selection options, and no parentheses to indicate simple iteration. Then the command processor may choose to skip calling the command preprocessor and may call the invoker modules directly instead.

The Listener

The listener is the crux of the command processor. It inputs commands from the user by calling the command line reader, passes them from the abbreviation processor, and executes them by calling the command processor.

An invocation of the listener is called a command level. Initially, after you log in, you are placed at command level 1, which is the first invocation of the listener on the stack. As you enter commands at level 1, the listener executes them.

The listener also establishes an on-unit for the ANY$ condition. Therefore, if you type Control-P or if a program you execute encounters an error such as an illegal instruction or an access violation, the first invocation of the listener catches the condition signaled (QUIT$, ILLEGAL_INST$, or ACCESS_VIOLATION$).

The default on-unit, named DF_UNIT_, responds to many of these conditions by displaying a useful message and then invoking the listener. This second invocation of the listener does not supersede the first; instead, the first invocation remains suspended (due to the interruption). The second invocation of the listener is command level 2 for the user.

For the most part, commands issued at command level 2 do not affect the program that was running at command level 1. (Exceptions primarily concern static mode programs, which do interfere with static mode programs at lower levels.) Therefore, when you have resolved the error, you may return to command level 1 and continue execution of the program at that level by issuing the START command. Alternatively, you may return to command level 1 and abort execution of the program at that level by issuing the RELEASE_LEVEL command.

The primary purpose for this creation of new command levels is to allow you to trace the cause of an interruption or program error while the stack history of the program at command level 1 is still maintained. For example, you can use the DUMP_STACK command at command level 2 to display stack frames for all procedures invoked between the signaling of the condition that caused the invocation of the default on-unit and the bottom of the stack. (The display begins with the most recently created condition frame still on the stack.)

An added benefit of this layering of command levels is that you can interrupt a command, execute a different command (or sequence of commands), and then continue the interrupted command. However, such cases may involve interactions that might prevent you from continuing the interrupted command successfully.

Each invocation of the listener is aware of its invocation number, which is also the command level number for the user. If this value exceeds the maximum number of allotted command levels, the listener displays an informative message and allows only mini-commands to be entered. This state is called mini-command level. The limited set of mini-commands are all internal commands, and they all either display information on resources (current usage or limits) or reduce resources used (by releasing command levels, removing programs from memory, logging out, and so on). None of the mini-commands allow you to invoke a program or to acquire an additional command level. In addition, Control-P no longer causes the generation of an additional command level; instead, it displays an error message and returns you to mini-command level.

The Command Prompter

The command prompter is called by the listener to display a prompt on the user's terminal so that the user knows that the system is ready and waiting for another command. The listener selects a ready, warning, or error prompt when it calls the prompter. (OK, is the default ready and warning prompt, ER! is the default error prompt.) The chosen prompt informs the user whether the most recently issued command completed successfully.

The command prompter can also display long prompts, which provide more information than the normal brief prompts. Information includes the time of day, incremental CPU and I/O time used by the user, and the command level number.

The RDY command determines what prompts will be available for display; used with no arguments, it calls the prompter and requests display of a long ready prompt.

The Command Line Reader

The command line reader is called by the listener to read a command line from the command input source, which may be the user's terminal, a command input file, or a &DATA block in a CPL program. The command line reader performs terminal erase and kill key processing for interactive input.

The Abbreviation Processor

After the listener has read a command line from the command line reader, the listener may pass the command line through the abbreviation processor. It will do this if the input source is the user's terminal and if the user and the System Administrator have both enabled abbreviation processing.

Note

If the input source is a CPL program with abbreviation processing enabled, then the CPL processor will pass the command line through the abbreviation processor.

The abbreviation processor is a time-saving device for entering commands. It expands short character sequences in a command line and replaces them with longer, more complex sequences. It then returns the resulting command line to the listener for further processing. Using the ABBREV command, users define the particular expansions they require, and enable and disable abbreviation processing.

The Command Processor

The listener then takes the command line and calls the command processor with it. The command processor is another critical part of the command environment.

The name of the command processor is CP$, which is a subroutine that may be invoked by user programs. In fact, CP$ simply invokes STD$CP, which is the standard command processor for PRIMOS. For the rest of this chapter, the term "command processor" will refer to STD$CP.

The command processor controls all of the remaining processing of the command line up to the point where the target program is actually invoked. (When this processing is complete, the command processor executes the resulting command(s). It then returns to the listener, which then prompts the user for the next command.)

The first step performed by the command processor is the splitting up of the command line into separate command lines if the command separator (;) has been used in the command line. The command processor then processes each of these split command lines, one at a time.

For each split command line, the command processor passes the command line through the expression evaluator. It then removes any null tokens in the command line. (A null token is a token consisting only of two single quotes.)

Then the command processor parses the first token in the command line. This is the command name. It uses the command name to determine the type of program being invoked and to decide which command processing features are to be inhibited and which are to be enabled for that particular command. It uses this information next, when it passes the command line to the command preprocessor along with the invoker module appropriate to the type of program being invoked.

The command preprocessor handles the task of performing any iteration (simple iteration, treewalking, wildcarding) and other preprocessing (name generation, object type selection, and so on); it also calls the appropriate invoker for the program each time it processes an iteration of the command line.

The Expression Evaluator

The expression evaluator is called by the standard command processor to resolve references to functions and CPL variables found in the command line. These references are signaled by the presence of brackets, [ ] (function references), and percent signs, % (variable references).

The evaluator replaces all variable references in the command line with the actual values of the variables. It then invokes the functions referenced in the command line, replacing their references with the

values returned by the functions. It does this only once; if the replacement value for a function contains the [,], or % characters, or if the replacement value for a variable contains the % characters, the expression evaluator does not evaluate them again; it leaves them as they are. (A special function called RESCAN may be used to reevaluate such references when desired.)

Once all of the references have been replaced with their values, the command processor determines the name of the command being invoked, the type of program being invoked (internal, EPF, CPL, or static-mode), and the command preprocessing features that apply to this program. It uses the command features decoder to determine the latter information.

The Command Features Decoder

Once the command processor knows the name of the command being invoked and the type of program it is, it uses the command features decoder to determine which command processing features are to be inhibited and enabled for the command.

The command features decoder handles program EPFs by reading out of the EPF itself the information on command processing features. This information is placed in the EPF by BIND when it generates an EPF. To change this information from its default settings (and thus to request non-default processing of features), programmers use BIND subcommands such as WILDCARD, NAMEGENPOS, NO_TREEWALK, and so on.

The decoder handles an internal command by reading the information out of the internal commands table in PRIMOS.

All CPL programs have the same features inhibited and enabled, as described earlier in this chapter. Thus, the decoder handles CPL programs very easily.

The decoder handles a static-mode program almost as easily as a CPL program. The features for a static-mode program depend upon whether it has an NX$ or NW$ prefix, or no such prefix, as described earlier in this chapter.

The command processor passes the information thus brought forth by the decoder to the command preprocessor.

The Command Preprocessor

The command processor now calls the command preprocessor with:

- The command line as it currently stands
- Information on command processing features

- The appropriate invoker module for the command (based on the type of program being invoked)

The command preprocessor searches the command line for iteration specifications and special options, as selected by the command processing features. Any iteration, object type selection, user verification, and name generation is then performed by the command preprocessor by generating multiple copies of the command line.

As each copy of the command line is generated and, optionally, approved by the user, the command preprocessor calls the invoker module with the copy of the command line. The invoker module calls the target program.

The Program Invokers

There are four separate program invoker modules, one for each type of program. Each module uses the same calling sequence which includes the command line, the severity code for the program, the command state structure, the command flags, and the returned function value pointer. (Chapter 3 describes this information in detail.)

The appropriate module then invokes the target program (or internal PRIMOS subroutine) with the same information. For example, the EPF invoker module calls the EPF$INVK subroutine, whereas the CPL invoker calls the CPL interpreter. (Because CPL consists of PRIMOS command lines, the CPL interpreter invokes the command processor recursively to execute PRIMOS commands specified in the CPL program. If the CPL program invokes another CPL program, the recursively invoked command processor winds up invoking the CPL interpreter again, recursively, to execute the second CPL program.)

When the target program completes, it returns to its invoker module. The invoker module then returns to the command preprocessor, which either proceeds to the next iteration of the command line or, when finished, returns to the command processor. When the command preprocessor returns to the command processor, execution of the original command line is complete, so the command processor returns to the listener. The listener then issues a prompt indicating the severity level for the entire command line and awaits a new command.

The Default On-unit

The listener establishes the default on-unit, named DF_UNIT_, as the handler for the wildcard ANY$ condition. Any conditions signaled for the process that are not handled by the running program reach the default on-unit, which performs appropriate default actions for the condition. For example, the default behavior upon receipt of the QUIT$ condition is to clear terminal buffers, display the QUIT message on the user's terminal, suspend command input, and invoke the listener again to create a new command level.

The default on-unit is not reserved for unusual circumstances. In fact, it is a crucial part of the command environment because it is used by many of the modules in the command environment to communicate between command levels.

For example, when a static-mode program invoked on level 1 is overwritten by a static-mode program invoked on level 2, the static-mode invoker signals a condition that is caught by the default on-unit for the level 2 invocation of the listener. The condition tells the default on-unit that a START or REENTER from that level to the level below should not be allowed if a static-mode program is in use there. The default on-unit records this information for use by owning the current invocation of the listener (in this example, the invocation for level 2). Then, if the listener is instructed by the user (via a START or REENTER command) to attempt to continue executing the program on level 1, the listener knows to reject the attempt because the program on level 1 has probably been overwritten.

The default on-unit also performs default handling of arithmetic overflow and other related error conditions, as appropriate. For example, if the user types START after the default on-unit has displayed information on an arithmetic overflow condition, the default on-unit sets the result of the arithmetic operation to an appropriate value before continuing execution of the program. Otherwise, the program would simply abort again when restarted.

Because of the nature of the condition-signaling mechanism, any of the actions of the default on-unit may be overridden by a user program if it establishes its own on-units for conditions.

However, because some conditions are used by the command environment as a form of internal communications, and because many of these special conditions are handled by the default on-unit, no user program should catch an unrecognized condition without continuing the signal.

The default on-unit is not the only module in the command environment that catches internal signals. The listener and the command processor are examples of modules that catch signals used for internal communication. The default on-unit may, in fact, signal some of these conditions caught by other modules in the command environment as a result of catching some other condition. For example, when the default on-unit catches the condition LIBRARY_IO_ERR$, which indicates an error in using language I/O, it displays appropriate error messages and then signals the condition STOP$. The STOP$ signal is caught by the command processor and recognized as a program termination.

2
Command Line Processing

This chapter explains how features of the command processor interact with program EPFs.

The PRIMOS Command Processor, invoked by calling CP$, performs several types of command processing before actually invoking the desired command. These types of processing, in the order performed, are:

1. Handling the command separator character (;)
2. Evaluation of command function and CPL variable references
3. Removal of null tokens (tokens containing only '')
4. Determination of command name
5. Determination of command type (internal, EPF .RUN program, static-mode .SAVE program, .CPL program, or static-mode program without suffix)
6. Determination of command iteration features enabled by command
7. Expansion of simple iteration
8. Expansion of treewalking
9. Expansion of wildcard specifications
10. Expansion of name generation patterns

After the above steps are performed, the target command or program is invoked with the final command line; (or, if at least one of the forms of iteration was used with many somewhat different copies of the command line).

For an example of the step-by-step processing of a command line containing several command line features (command separators, abbreviation expansion, iteration, treewalking, and wildcarding), as well as for further discussion of command line features from the user's point of view, see the PRIMOS Commands Reference Guide.

There are many different ways to inhibit most of the 10 steps of command line processing listed above. (Steps 4 and 5 must always be performed to execute the command.) These are:

- Use of the tilde (~) as the first character in the command line; this suppresses expansion, and thus inhibits Steps 1, 3, and 6 through 10
- Invocation of a function (by setting the Function-Call bit in the flags argument of CP$), which inhibits Steps 1, 3, and 6 through 10
- Disabling function and variable evaluation (by setting the Inhibit-Evaluation bit in the flags argument of CP$), which inhibits Step 2
- Invocation of the ABBREV (or AB) command, which inhibits any remaining activity in Step 1, and which inhibits Steps 8 through 10
- Invocation of a CPL program, which inhibits Steps 8 through 10
- Invocation of a static-mode program with the NX$ prefix, which inhibits Steps 7 through 10
- Invocation of a static-mode program with the NW$ prefix, which inhibits Steps 8 through 10
- Invocation of a program EPF built using the NO_ITERATION subcommand of BIND, which inhibits Step 7
- Invocation of a program EPF built using the NO_TREEWALK subcommand of BIND, which inhibits Step 8
- Invocation of a program EPF built using the NO_WILDCARD subcommand of BIND, which inhibits Step 9
- Invocation of a program EPF built using the NO_GENERATION subcommand of BIND, which inhibits Step 10

In addition, a particular step is inhibited if it is keyed to a character or to a sequence of characters (such as ; for command separation or () for iteration) and the key is either not present on

the command line or is present only within single quotes. For example, the following two command lines execute with Step 1 inhibited:

```
TYPE Compiling main program.

TYPE Compiling subroutine PLOTXY';' language is F77.
```

The rest of this chapter focuses on each of the above 10 steps in more detail — in particular, the character sequence keys that ignite each step are listed and explained.

STEP 1: HANDLING THE COMMAND SEPARATOR CHARACTER

The command separator character (;) is the key for Step 1. If present and unquoted, Step 1 causes the original command line to be split up into two or more command lines at each occurrence of the semicolon (;). Each of these separate command lines is then passed through the remaining steps, one by one, in the order in which it appeared in the original command line.

Because each command line is treated separately, each may inhibit or enable different combinations of Steps 7 through 10.

However, there exists a special case: the ABBREV command (and its abbreviation, AB). After Step 5, if the command processor sees that it is evaluating an internal command, it checks whether the command is the ABBREV or AB command. If it is, the command processor treats any remaining split command lines (following semicolons after the ABBREV command) as part of the ABBREV command line, along with the semicolons. Then it passes the assembled command line through Steps 2 through 7 before executing it.

In other words, the command line

```
TYPE HELLO;ABBREV -STATUS
```

displays the word HELLO followed by the output of the ABBREV -STATUS command, whereas the command line

```
ABBREV -STATUS;TYPE HELLO
```

produces the following error message:

```
Control argument "-STATUS;TYPE" not implemented. (abbrev)
ER!
```

The purpose of this exception for ABBREV is to allow users to create abbreviations that contain semicolons. Note, however, that function and variable references may still be evaluated (unless Step 2 is inhibited).

STEP 2: EVALUATION OF FUNCTION AND CPL VARIABLE REFERENCES

If the command line contains the characters [or %, the command processor performs the evaluation of function and CPL variable references.

Whereas variable references are simply replaced by actual string values, function references are replaced by calling the command processor recursively to invoke the desired function and then substituting the returned value. (Only function calls of external programs are counted against your maximum command environment breadth.)

STEP 3: REMOVAL OF NULL TOKENS

If the command line contains any single quotes ('), the command processor removes null tokens (tokens containing only '') in this step. For example, if the command line reads

```
COPY A B '' ''
```

then the command line after this step becomes:

```
COPY A B
```

This step is necessary because command preprocessing performed up to this point, such as abbreviation processing (performed by the listener), and function and variable evaluation, may result in null tokens. Such null tokens might not be handled correctly by the target program.

STEP 4: DETERMINATION OF COMMAND NAME

At this step, the first token of the command line becomes the command name. The command may be an internal command or one of several types of external command, as determined in the next step.

STEP 5: DETERMINATION OF COMMAND TYPE

In this step, the command processor searches for the command in its list of internal PRIMOS commands. If the command is present in the list, the command is an internal command. Otherwise, the command processor searches the CMDNC0 directory for the command as described in Chapter 1. The suffix on the file found tells the command processor what type of command is to be invoked.

If the command is RESUME, the command processor treats the entryname portion of the pathname following the RESUME token as the actual command name.

STEP 6: DETERMINATION OF COMMAND ITERATION FEATURES

Depending upon the command type, the command processor determines which of the remaining steps are to be inhibited. Internal commands and program EPFs selectively enable or disable each of the remaining steps according to information in the internal command table (for internal commands) or in the .RUN file (for EPFs); all CPL programs inhibit Steps 7 and 8 but enable Steps 9 and 10.

Static-mode programs inhibit or enable the remaining steps based on the command name. If the name begins with NX$, all of the remaining steps are inhibited. If the name begins with NW$, only Steps 8 through 10 are inhibited. Otherwise, all steps are enabled.

The RESUME command is treated specially, as described in the previous step. The command iteration features for the RESUME command are not determined by the internal command table entry for RESUME but are determined instead by the program that is the target of the RESUME command. In addition, the name generation pattern is considered to be the token following the name of the program being invoked, rather than the name itself.

If the program being invoked is CMDNC0>SEG, the name generation pattern is considered to be the token following the pathname that follows the SEG command or the RESUME CMDNC0>SEG command, rather than the pathname of the .SEG file itself.

STEP 7: EXPANSION OF SIMPLE ITERATION

If the command line contains parentheses, that is, (and), simple iteration is performed. For each iteration, a new command line is built that contains no parentheses; the command processor passes this new command line through the remaining steps before executing.

STEP 8: EXPANSION OF TREEWALKING

If the command line contains a valid pathname with a directory portion that contains wildcard characters, (@ + or ^), the command processor honors the following command line options:

-WALK_FROM (-WLKFM)
-WALK_TO (-WLKTO)
-BOTTOM_UP (-BOTUP)

If Step 8 is enabled but no treewalk specification appears on the command line, these options are ignored and are not passed to the target program.

As the command processor matches each directory to the treewalk specification, it passes the resulting command lines through Step 9.

STEP 9: EXPANSION OF WILDCARD SPECIFICATIONS

If the command line contains a pathname with an entryname portion that contains a wildcard character, (@ + or ^), and if either the directory portion of the pathname contains no wildcard character or Step 8 is enabled, the command processor performs wildcard expansion honoring the following command line options:

-BEFORE (-BF)
-MODIFIED_BEFORE (-MDB)
-AFTER (-AF)
-MODIFIED_AFTER (-MDA)
-BACKEDUP_BEFORE (-BKB)
-BACKEDUP_AFTER (-BKA)
-RBF
-FILE
-DIRECTORY (-DIR)
-SEGMENT_DIRECTORY (-SEGDIR)
-ACCESS_CATEGORY (-ACAT)
-VERIFY (-VFY)
-NO_VERIFY (-NVFY)

If this step is enabled but no wildcard specification is on the command line, these options are ignored and are not passed to the target program.

The default options depend upon the command name and command type. For static-mode programs, the defaults are:

-FILE -DIR -SEGDIR -ACAT -NO_VERIFY

For internal commands and EPFs, the defaults depend upon the command name. The default for an internal command resides in the internal commands table in PRIMOS, while the defaults for an EPF are set during the BIND session that created the EPF. CPL programs have no applicable defaults because they always inhibit Step 9.

As the command processor compiles a list of items that match the given wildcard specification, it may ask the user to verify (or approve action on) each item:

- If -VERIFY was specified, it will request verification.
- If -NO_VERIFY was specified, it will not request verification.
- If neither -VERIFY nor -NO_VERIFY was specified, it either does or does not request verification, depending on the default for that particular command.

As the user affirms each matching object, or as each matching object is found (if no verification is taking place), the command processor builds a command line for each object. When the list of objects has been compiled, the command processor passes each resulting command line through Step 10.

STEP 10: EXPANSION OF NAME GENERATION PATTERNS

If a token on the command line contains the name generation character (=), the command processor performs name generation. Name generation characters also include ^ and +, although = must be present for name generation to be performed.

The command processor analyzes the source pattern for the name generation. The source pattern is a particular token on the command line, typically the first argument, although internal commands and EPFs may select subsequent arguments as their name generation source patterns.

Then, the command processor combines the source pattern with each token containing = to replace the tokens with actual names. After each command line is constructed, the command processor invokes the target command or program.

INVOCATION

The final step is invocation of the target command or program. There are four invocation modules:

- The internal-command invoker
- The EPF invoker
- The CPL-program invoker
- The static-mode-program invoker

Each of these invokers calls the target command or program, and each regains control when the command or program returns. (Static-mode programs are specially handled when they call EXIT so that they return to the static-mode-program invoker.)

If some form of interruption occurs, causing a new command level to be obtained, the iteration at the original level is only suspended, not terminated. Continuing the interrupted program resumes iteration where it left off.

Moreover, if iteration is in progress at command level 1, and the user types Control-P to reach command level 2, the user may issue another command that performs iteration without disturbing the suspended iteration at command level 1. After the second command has finished, the user may use the START command to continue with the iteration begun at command level 1.

However, if the user releases the original level, moving down to a previous level; or, if the user releases to the original level, thus releasing the target program and the invocation of the command processor for that program, all of the iteration is terminated.

3

Program EPF Calling Sequence

The main entrypoint of a program EPF is invoked by the command environment with a standard calling sequence. This calling sequence consists of five arguments:

1. The command line, supplied by the invoker
2. The command status, set by the invoked program to indicate its level of success to the invoker
3. Information on the command processing state, supplied by the invoker
4. A flag indicating whether the invoker desires a return value — that is, whether the invoker is treating the invoked program as a command function
5. A pointer, set by the invoked program to point to the returned value structure

The complete calling sequence is illustrated near the end of this chapter; however, very few programs need all the information and arguments provided by the command environment. In fact, most programs need accept only two or fewer arguments.

The invoker is always the EPF$INVK subroutine. EPF$INVK may be called directly by user programs, by the EPF$RUN subroutine, or by the CP$ subroutine. EPF$RUN itself is called directly by user programs. CP$ is also callable by user programs, and is called by PRIMOS to execute a command.

TYPES OF CALLING SEQUENCES

There are five types of program EPF calling sequences, with various levels of complexity. They are:

1. The program calling sequence, which takes no command line and which returns no information

2. The command calling sequence, which accepts a command line and which returns a severity code

3. The command function calling sequence, which accepts a command line and which returns both a severity code and a pointer to the returned function value

4. The detailed command calling sequence, an extended form of the command calling sequence that accepts detailed command processing information

5. The complete calling sequence, which combines the command function calling sequence with the detailed command calling sequence

The remainder of this chapter describes each of the calling sequences listed above.

In all cases, the EPF$INVK subroutine passes either zero or five arguments to the EPF it invokes. It determines the number by examining the ECB of the EPF's main subroutine. If the ECB shows that the subroutine accepts no arguments, EPF$INVK passes none (thus using the program calling sequence for the invocation). Otherwise, EPF$INVK passes all five arguments to the invoked EPF; the EPF itself decides how many of the arguments to accept. Any arguments it does not accept, it ignores. (The PCL instruction, which performs procedure calls on Prime systems, handles this situation properly.) Alternatively, the main subroutine of the invoked EPF may accept all five arguments but choose to ignore some or all of them.

Except for the program calling sequence, therefore, the five types of calling sequence listed above are differentiated not by the actions of EPF$INVK but by the number of arguments that the main subroutine has been designed to accept. This chapter differentiates and describes them to simplify your job when you construct the main program of an EPF. By looking at the descriptions of the functionality each calling sequence provides, you can decide what kind of program you are writing and then choose the calling sequence that best suits your program.

PROGRAM CALLING SEQUENCE

The program calling sequence is the simplest calling sequence because it accepts no arguments. Any command line passed to such a program is ignored; no severity code is returned, so a severity code of 0 is assumed by the invoker; if the program is invoked as a command function, no pointer to the returned value is returned.

The calling sequence is not illustrated, because it contains no input or output arguments.

A program whose main subroutine accepts no arguments may use the SETRC$ subroutine, described in the Subroutines Reference Guide, to return a severity code, even though it does not accept the severity code argument in its main subroutine. This feature eases the conversion to an EPF of an existing static-mode program that uses SETRC$.

COMMAND CALLING SEQUENCE

The command calling sequence is used for programs that accept command line arguments and options and that return a severity code.

Arguments in the Command Calling Sequence

The command calling sequence is the simplest calling sequence that accepts arguments. It accepts two arguments:

1. The command line, an input-only argument
2. The severity code, an output-only argument

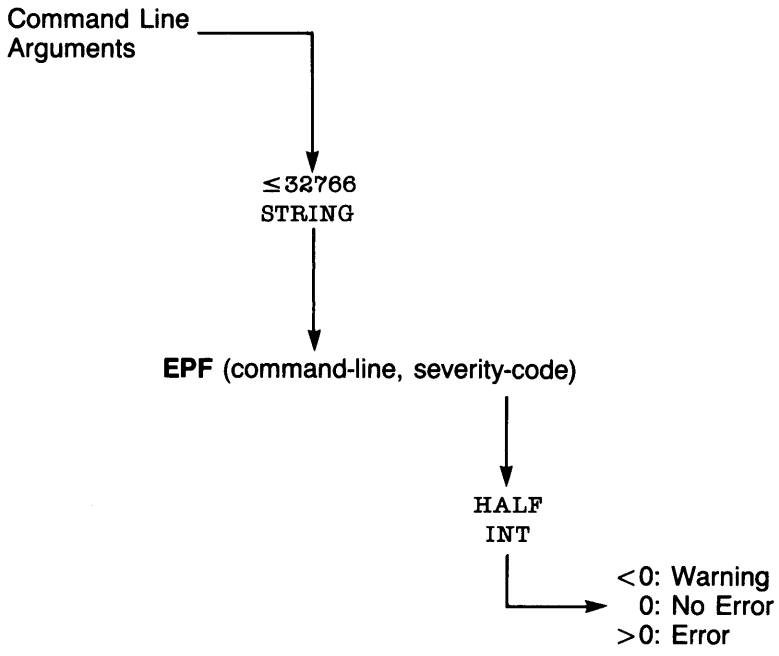
If a program that accepts only these two arguments is invoked as a command function, no pointer to the returned value is returned.

Figure 3-1 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

Command Line: The length of the command line can be a maximum of 32,766 characters. Your program may limit the length to any value it chooses. Practical limits depend on the source of the command line. For example, the limit on the length of a command line entered by an interactive user, or from a command input file, is 160 characters, whereas the limit on the length of a command line in a CPL program is 1024 characters.

If your program is passed a command line that is longer than it can handle, it should use the error code E$TRCL both as a severity code and as an error code to ERRPR$ to indicate that the command line has been truncated. If your program aborts due to this condition, then a

Command Calling Sequence



Command Calling Sequence
Figure 3-1

truncated command line is an error; therefore, your program should return E$TRCL, a positive value, as the severity code. If your program continues processing, but uses a truncated form of the command line, your program should return -E$TRCL, a negative value, as the severity code (unless a positive error code is required for other reasons) to indicate a warning condition.

In PL/I-G, you can use the LENGTH built-in function to check whether the length of the command line is greater than your program supports, even if you have declared the command line to be the maximum size your program supports. In FORTRAN and other languages, you can compare the first halfword of the command line argument, which is the actual length of the command line, to the maximum length your program supports.

If your program does not accept a null command line, it should use the E$NCOM error code to indicate that it has been passed a null command line. In addition, you may wish to have your program display usage information when passed a null command line; this is what many Prime-supplied programs, such as SPOOL and JOB, do with a null command line. Even if your program does display usage information, it should still return E$NCOM, a positive value, as the severity code to indicate an error.

Other error codes your program may wish to return as either positive values (to indicate errors) or as negative values (to indicate warnings), and which your program may also wish to use when calling ERRPR$ to display warning messages, are:

Error Code	Used For
E$BPAR	Invalid numeric arguments — arguments where a number was expected but some other argument was supplied
E$BNAM	Invalid file system objectname arguments
E$NMLG	Overly long names, such as a file system objectname that is more than 32 characters long
E$ITRE	Invalid pathnames
E$CMND	Invalid command formats, such as the use of an option when no options are allowed, or the use of command line arguments when no command line arguments are allowed
E$BARG	Invalid arguments, such as the use of an unrecognized option, or the use of a name or number when an option was expected
E$IVCM	Invalid usage of a command, such as a combination of options and arguments that is not permitted or that does not make sense

E$MISA	Missing arguments, such as when a number, name, or option that is required is not provided on the command line

All standard PRIMOS error codes, including those shown above, are listed along with their numeric equivalents, messages, and descriptions, in Volume 0 of this series.

Severity Code: Your program should set the severity code to an appropriate value before returning from its main subroutine. The meaning of a severity code depends on whether it is negative, zero, or positive. The magnitude of the severity code is not defined by PRIMOS; however, your program should have documentation that describes the different severity codes it may return and what they mean. Typically, standard PRIMOS error codes, listed in Volume 0 of this series, are used for severity codes; to indicate warning conditions, the negated values of standard PRIMOS error codes are often used.

COMMAND FUNCTION CALLING SEQUENCE

The command function calling sequence is used when the program expects to be invoked as a command function. It may or may not expect command line arguments and options, and it may or may not return a severity code. Such a program constructs a returned value — that is, a text string that can be substituted on the command line for the function reference that invoked the program. It then returns a pointer to the structure that contains that returned value.

The steps a command function performs are:

1. Accept five arguments in the main entrypoint calling sequence.
2. Determine the string value to be returned to the calling program.
3. Allocate memory for the string value to be returned.
4. Copy the string value into the allocated memory.
5. Store the pointer to the allocated memory into the pointer passed in the calling sequence of the main entrypoint.
6. Return to the calling program.

Step 1, accepting five arguments in the main entrypoint, is described below in the section entitled Arguments in the Command Function Calling Sequence. Step 2, determining the value to be returned, depends on the purpose of your program. Steps 3 and 4 are usually combined into one step by calling the ALS$RA subroutine, described below in the section entitled The ALS$RA Subroutine. Alternatively, they may be performed

separately by calling the ALC$RA subroutine and then copying the string value afterwards. Typically, only programs written in PL/I-G or PMA perform Steps 3 and 4 separately.

Step 5 is often performed implicitly during Step 3 if ALS$RA or ALC$RA is passed the same variable that was accepted in the calling sequence of the main entrypoint; otherwise, your command function must explicitly set the rtn-fcn-ptr variable passed to it in the calling sequence of the main entrypoint so that it points to the structure allocated by ALS$RA or ALC$RA.

Step 6 is performed in the same way for functions as for other types of programs. Your program should set the returned severity code to an appropriate value before returning.

After the next three sections, a section entitled Sample Command Functions presents two simple sample command functions.

Arguments in the Command Function Calling Sequence

The main subroutine of a command function accepts five arguments:

1. The command line, an input-only argument
2. The severity code, an output-only argument
3. An input-only argument, which may be ignored by most command functions
4. The invocation form bit, an input-only argument
5. The returned value pointer, an output-only argument

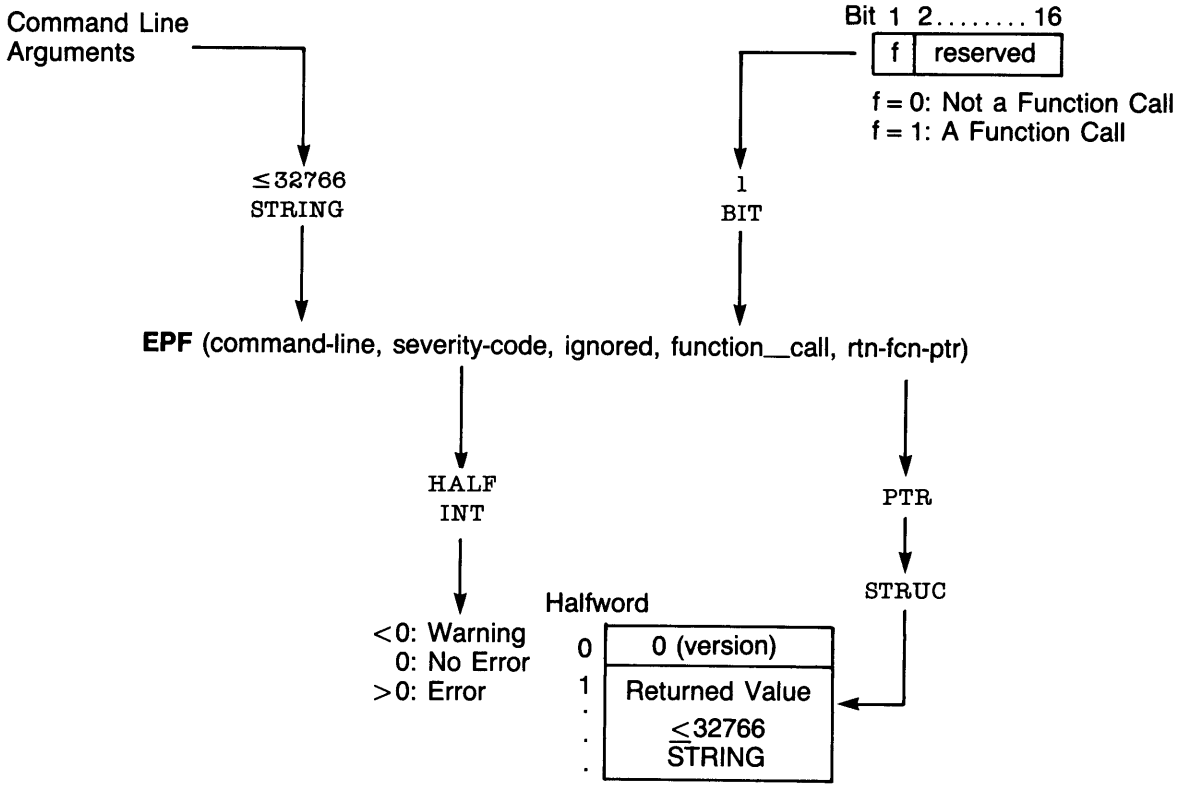
Figure 3-2 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

Command Line: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the command line. That information applies to command functions as well.

Severity Code: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the severity code. That information applies to command functions as well.

Ignored: The information passed to a program in the third argument may be ignored by most command functions. It is described in the next section, entitled DETAILED COMMAND CALLING SEQUENCE.

Command Function Calling Sequence



Command Function Calling Sequence
Figure 3-2

Invocation Form: The form of program invocation is a bit that indicates whether the program is being invoked as a command function or as a normal command. When set (1), function-call indicates that the invoker expects the program to set rtn-fcn-ptr to point to a structure containing the returned value of the function. When reset (0), function-call indicates that the invoker does not expect the program to set rtn-fcn-ptr at all, and that in fact the invoker may not have supplied the rtn-fcn-ptr argument.

> Caution
>
> Under no circumstances should your program set rtn-fcn-ptr when function-call is reset (0), nor should your program allocate storage for the returned value. When function-call is reset (0), the fifth argument, rtn-fcn-ptr, may not be passed to your program, and any attempt that your program makes to set it may therefore result in a POINTER_FAULT$ error condition being signaled. If the fifth argument is passed, but function-call is reset (0), then your program may succeed at setting rtn-fcn-ptr, but the invoking program will not expect it to point to the returned structure, and will therefore not deallocate the memory used by the structure.

Returned Value Pointer: If your program has been invoked with the function-call bit of the calling sequence set (1), then the invoking program expects your program to return a pointer to a structure that contains the returned value. The returned value is a text string of 0-32766 characters. The structure contains a version number (currently 0) as a HALF INT value and the returned value as a <=32766 STRING value.

After ALS$RA or ALC$RA returns a pointer to your program, your program must use the rtn-fcn-ptr argument to return that pointer to its invoker. The calling program will pass the pointer your program returns in rtn-fcn-ptr to the FRE$RA subroutine (described in Chapter 4), so that FRE$RA can free the storage allocated by ALS$RA or ALC$RA.

> Caution
>
> If your program does not use ALS$RA or ALC$RA to determine the rtn-fcn-ptr pointer, but uses instead a pointer constructed by other means, then when the calling program calls FRE$RA with the returned pointer, a fatal error will occur.

The ALS$RA Subroutine

The ALS$RA subroutine allocates sufficient memory to hold the supplied string value, copies the string value into the allocated memory, and returns the pointer to the allocated memory for use by the program that

invoked the command function. The calling sequence for ALS$RA is illustrated in Figure 3-3.

Your program passes ALS$RA the string value to be returned in value and its size, in characters, in value-size. ALS$RA allocates sufficient memory (at least (value-size+5)/2 halfwords) to hold the string value; sets the first halfword of the allocated memory to 0 to indicate a version 0 returned value structure; stores the length of the string in value-size into the second halfword of the allocated memory; copies the string in value into the allocated memory starting with the third halfword; and returns a pointer to the first halfword of the allocated memory in rtn-fcn-ptr.

After calling this subroutine, your program needs only to ensure that the pointer returned by ALS$RA is returned by the main entrypoint of your program to the calling program. Your program ensures this by storing the pointer into the rtn-fcn-ptr argument of its main entrypoint. Then, your program simply returns to its invoker. The invoking program is responsible for deallocating the memory allocated by ALS$RA.

The ALC$RA Subroutine

The ALC$RA subroutine is similar to the ALS$RA subroutine, except that it does not copy the string value into the allocated memory. It leaves this task to your program, the command function.

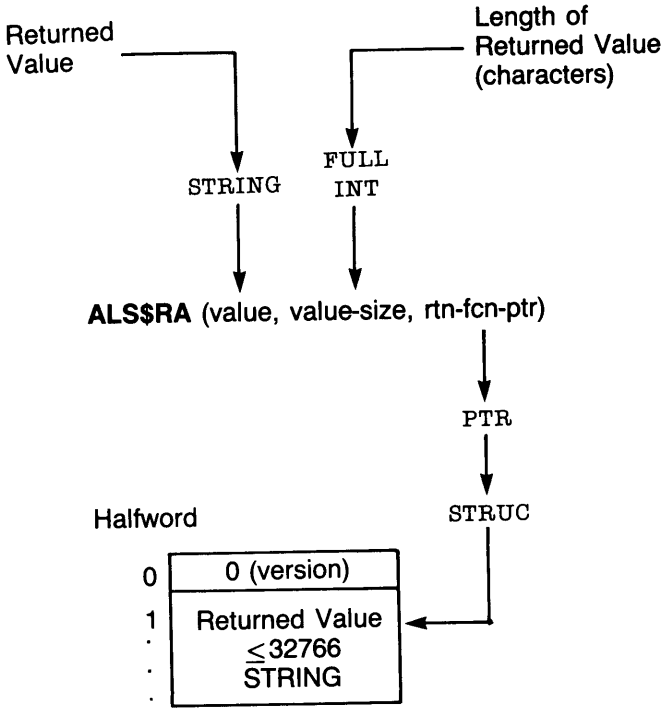
The ALC$RA subroutine allocates sufficient memory to hold a string value of the specified length and returns the pointer to the allocated memory for use by your program. The calling sequence for ALC$RA is illustrated in Figure 3-4.

Your program passes the number of halfwords to be allocated in halfwords. This value should be at least (value-size+5)/2, where value-size is the length of the string value to be returned. ALC$RA allocates the requested number of halfwords to hold the string value, and returns a pointer to the first halfword of the allocated memory in rtn-fcn-ptr.

After calling this subroutine, your program must set the first halfword of the allocated memory to 0 to indicate a version 0 returned value structure; set the second halfword of the allocated memory to the length of the string value in characters; then copy the string value into the allocated memory starting at the third halfword of the allocated memory. Your program must use the rtn-fcn-ptr pointer to perform these tasks; therefore, only programs written in PL/I-G or PMA are likely to use this interface.

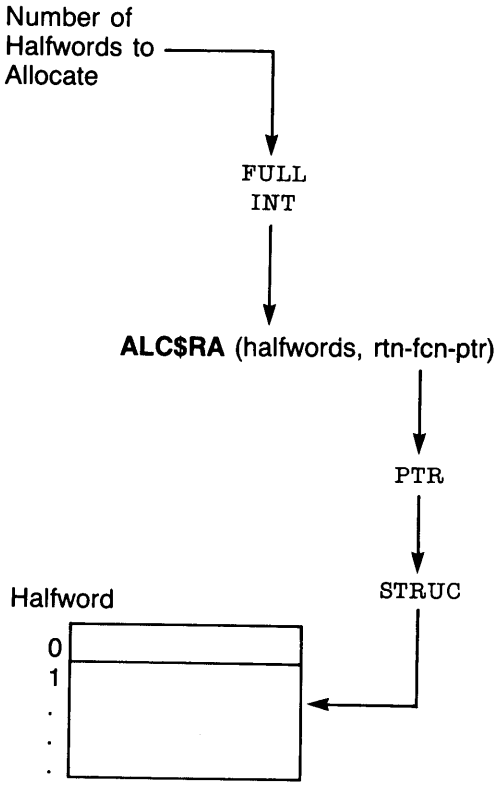
After copying the string value into the allocated memory, your program must store the pointer returned by ALC$RA into the rtn-fcn-ptr argument of your program's main entrypoint, in order to ensure that the pointer is returned to the calling program. Then your program simply returns

Allocate and Set Returned Function Value



The ALS$RA Subroutine
Figure 3-3

Allocate Space for Returned Function Value



The ALC$RA Subroutine
Figure 3-4

to its invoker. The invoking program is responsible for deallocating the memory allocated by ALC$RA.

Sample Command Functions

The first sample program is a FORTRAN program that returns the usernumber of the user invoking the program.

```
      SUBROUTINE USRNUM(COMLIN,CODE,IGN,FUNC,RINPTR)
      INTEGER*2 COMLIN(1),CODE,IGN,FUNC
      INTEGER*4 RINPTR(2)
C
$INSERT SYSCOM>ERRD.INS.FTN
$INSERT SYSCOM>KEYS.INS.FTN
C
      INTEGER*2
     &   U,           /* User number; later, units digit of U.
     &   TIMARR(12),  /* TIMDAT array.
     &   STR(2),      /* String value containing user number.
     &   STRLEN,      /* Number of characters in STRLEN.
     &   H,           /* Hundreds digit of U.
     &   T            /* Tens digit of U.
C
C Make sure we have no command line.
C
      IF (COMLIN(1).EQ.0) GO TO 10
C
C Reject attempted use of command line.
C
      CODE=E$IVCM                 /* Invalid command error.
      IF (AND(FUNC,:100000).EQ.0)  /* Invoked as command?
     &   CALL ERRPR$(K$IRTN,CODE,'No command line accepted',24,
     &   'USERNUMBER',10)
      RETURN                      /* Return to invoker.
C
10    CALL TIMDAT(TIMARR,12)      /* Get user number in TIMARR(12).
      U=TIMARR(12)                /* For ease of access.
      IF (U.GT.9) GO TO 20        /* More than one digit?
      STR(1)=LS(U,8)+'0 '         /* Convert to single-digit ASCII.
      STRLEN=1                    /* Set to 1 digit.
      GO TO 100
C
20    H=U/100                     /* Get hundreds digit.
      U=U-H*100                   /* Get last two digits.
      T=U/10                      /* Get tens digits.
      U=U-T*10                    /* Get last digit.
      IF (H.NE.0) GO TO 30        /* Need three digits?
      STR(1)=LS(T,8)+U+'00'       /* No, make two digits into ASCII.
      STRLEN=2                    /* Indicate two digits.
      GO TO 100
C
```

```
30    STR(1)=LS(H,8)+T+'00'     /* Make three digits into ASCII.
      STR(2)=LS(U,8)+'0 '
      STRLEN=3                  /* Indicate three digits.
C
100   IF (AND(FUNC,:100000).NE.0) GO TO 200
C
C Not a function call; display user number.
C
      CALL TNOUA('Your user number is ',20)
      CALL TNOUA(STR,STRLEN)
      CALL TNOU('.',1)
      GO TO 300
C
C A function call; allocate and store user number.
C
200   CALL ALS$RA(STR,INTL(STRLEN),RTNPTR)
C
C Return to invoker.
C
300   CODE=0                    /* Success!
      RETURN
C
      END
```

The next sample program, written in PL/I-G, returns the username of the invoking user.

```
username: proc(comlin,code,ign,func,rtn_fcn_ptr);

dcl comlin char(32) var,  /* Must be null. */
    code fixed bin(15),   /* Severity code. */
    ign fixed bin(15),    /* Ignored. */
    func bit(1),          /* Set if function call. */
    rtn_fcn_ptr ptr;      /* Returned function value pointer. */

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl unam char(32) var;    /* Trimmed username. */

dcl 1 timarr,
      2 ignore (12) fixed bin(15), /* Ignore 12 halfwords. */
      2 user_name char(32);  /* The username. */

dcl 1 rtn_struc based(rtn_fcn_ptr),
      2 version fixed bin(15),
      2 value char(32) var;

dcl timdat entry(1,2 (12) fixed bin(15),2 char(32),fixed bin(15)),
    errpr$ entry(fixed bin(15),fixed bin(15),char(40),
                 fixed bin(15),char(8),fixed bin(15)),
    alc$ra entry(fixed bin(31),ptr),
```

```
      tnou entry(char(60),fixed bin(15)),
      tnoua entry(char(60),fixed bin(15));

   if comlin='' then
      do;  /* No command line. */
      call timdat(timarr,28);
      unam=trim(user_name,'11'b);
      if func then
         do;  /* Command function invocation. */
         call alc$ra(divide(length(unam)+5,2,15),rtn_fcn_ptr);
         rtn_struc.version=0;
         rtn_struc.value=unam;
         end;  /* if func */
      else
         do;  /* Command invocation. */
         call tnoua('Your user name is ',18);
         call tnoua((unam),length(unam));
         call tnou('.',1);
         end;
      code=0;  /* Success. */
      end;  /* if comlin='' */
   else
      do;  /* if comlin^='' */
      code=e$ivcm;
      if ^func then
         call errpr$(k$irtn,code,'No command line accepted',24,
                     'USERNAME',8);
      end;  /* if comlin^='' */

   end;  /* username: proc */
```

DETAILED COMMAND CALLING SEQUENCE

The detailed command calling sequence adds a third argument to the command calling sequence described earlier in this chapter. This third argument is a structure passed to the program EPF being invoked that includes the following information:

- The command name as entered by the user
- A pointer to CPL local variables, if appropriate
- Command preprocessing information

Typically, a program EPF uses only the portions of the structure that are applicable to the program. For example, if you wish your program to display the command name entered by the user (rather than the original name of your program) in error messages, you could have the main entrypoint of your program use only the command name as entered by the user and ignore the remainder of the structure.

The remainder of this section describes the information passed in the third argument of the program EPF calling sequence.

Arguments in the Detailed Command Calling Sequence

The detailed command calling sequence accepts three arguments:

1. The command line, an input-only argument
2. The severity code, an output-only argument
3. A structure containing command processing information, an input-only argument

If a program that accepts only these three arguments is invoked as a command function, no pointer to the returned value is returned.

Figure 3-5 illustrates the command calling sequence, where EPF is the main subroutine of the program EPF.

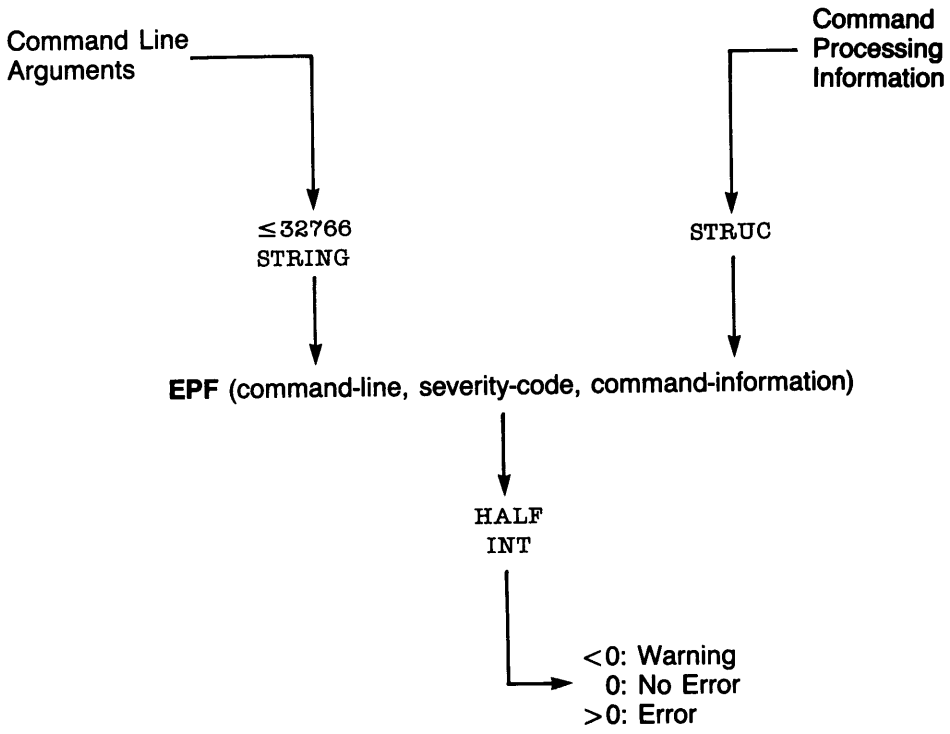
Command Line: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the command line. That information applies to detailed commands as well.

Severity Code: See the section earlier in this chapter entitled COMMAND CALLING SEQUENCE for information on the severity code. That information applies to detailed commands as well.

Command Processing Information Structure: Figure 3-6 illustrates the command processing information, which is described in detail in the next section.

Currently, two versions of the command processing information structure are defined. The first two fields, the command name and the version number, are always present. If version is 0, the remainder of the command processing information structure is undefined and should not be referenced; only halfwords 0-17 (0-21 octal) are defined for a version 0 structure. If version is 1, the entire structure is defined as shown; that is, halfwords 0-25 (0-31 octal) are defined. Future versions of the structure will have higher version numbers and may define extensions to version 1 of this structure; however, the content and meaning of halfwords 0-25 will remain the same.

Detailed Command Calling Sequence



Detailed Command Calling Sequence
Figure 3-5

Command Processing Information (Versions 0 and 1)

Halfword oct	Halfword dec								Halfword oct	Halfword dec
0 : 20	0 : 16	Command Name					≤32 STRING		0 : 16	0 : 20
21	17	Version (0 or 1)					HALF INT		17	21
22 23 24	18 19 20	CPL Local Variables Pointer					PTR		18 19 20	22 23 24
25	21	-DIR 1 BIT	-SEGDIR 1 BIT	-FILE 1 BIT	-ACAT 1 BIT	-RBF 1 BIT	Reserved	11 BIT	21	25
26	22	-VERIFY 1 BIT	-BOTUP 1 BIT	Reserved			14 BIT		22	26
27	23	-WALK_FROM Value					HALF INT		23	27
30	24	-WALK_TO Value					HALF INT		24	30
31	25	( ) 1 BIT	@ + 1 BIT	>@> >+> 1 BIT	Reserved		13 BIT		25	31

Note: For a version 0 structure, only halfwords 0-17 (0-21 octal) have defined values.

Command Processing Information
Figure 3-6

WARNING

Never store data into the command processing information structure for any purpose. Some calling programs may have declared only 18 halfwords of storage for a version 0 structure, representing halfwords 0-17, and any attempt to store beyond halfword offset 17 may corrupt memory. In addition, because the structure is an input argument to the program being invoked, the calling program may place the structure in memory that is protected against writing.

Your program should check the version number only if it needs to use information beyond halfword offset 17 (21 octal) into the command processing structure; and, in such a case, your program should check only that the version number is not 0 to ensure that the information being retrieved is valid. Do not reject version numbers higher than 1. However, if you choose, you may have your program reject version numbers that are negative, because such numbers probably indicate corrupted memory.

Command Processing Information

This section describes each field in the command processing information structure shown in Figure 3-6.

Command Name: The command name field contains the command name as specified by the user. The name will contain only the final element of a pathname; it may or may not include the .RUN suffix. Your program may use this name rather than the name designed for it in messages displayed to the terminal, or your program may reject attempts to invoke it with a name other than that which it was designed to have.

Typically, the command name is the same name specified during the BIND session that linked the program. However, if a user copies your program to a file with a different name and invokes the copy, or if the name of the file containing the program is changed (via CNAME for example), the command name will be different from the original name of the program.

Version: The version number field contains the version number of the command processing structure. Currently, version numbers 0 and 1 are defined as described above. Higher version numbers will be used if future versions of PRIMOS extend the command processing information structure. The following table lists the currently defined version numbers and the halfwords that are defined (have meaningful values) in a structure with each version number listed:

Version	Defined Halfwords
0	0-17
1	0-25

CPL Local Variables Pointer: The CPL Local Variables Pointer is provided if the calling program is either a CPL program or a program EPF provided with a CPL Local Variables Pointer (ultimately invoked by a CPL program).

Sometimes referred to as the vcb_ptr, for Variables Control Block pointer, this pointer is used only when the program EPF wishes to read or set a CPL variable that is local to the CPL program that invoked the program EPF. Typically, such programs are designed as command functions, and the CPL program uses the &SET_VAR directive, as in:

```
&SET_VAR MYVAR := [RESUME MYPROG]
```

However, a program that must reference more than one CPL variable must either be constrained to use only global variables (accessing them via the GV$GET and GV$SET subroutines) or must use the CPL Local Variables Pointer along with the LV$GET and LV$SET subroutines. A program constructed in the latter fashion might be invoked from a CPL program as follows:

```
RESUME MYPROG MYVAR OTHERVAR
```

Here, the MYPROG program accepts two variable names, MYVAR and OTHERVAR in this example, and accesses them using LV$GET and LV$SET, which are described (along with GV$GET and GV$SET) in the Subroutines Reference Guide.

The CPL Local Variables pointer is NULL() (7777/0) if the invoking program is not a CPL program, or if it is not a program EPF invoked by a CPL program (either directly or via other program EPFs). A valid CPL Local Variables pointer is generated only by the invocation of a CPL program, and is valid only while that program is active; only program EPFs invoked by the CPL program, and their descendants, may use the Local Variables pointer for that CPL program.

Note

For maximum flexibility, design your program so that it accepts either global variables — which have names beginning with a period (.) — or local variables — which have names not beginning with a period (.). Then, your program would call either GV$GET/GV$SET or LV$GET/LV$SET, depending on what type of variable name is supplied.

-DIRECTORY (-DIR) Bit: The -DIRECTORY bit is set if the command processor is matching file directories when checking wildcard-laden names. It does not necessarily mean that the file system object that is specified in the current invocation is a file directory.

-SEGMENT_DIRECTORY (-SEGDIR) Bit: The -SEGMENT_DIRECTORY bit is set if the command processor is matching segment directories when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is a segment directory.

-FILE Bit: The -FILE bit is set if the command processor is matching files when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is a file.

-ACCESS_CATEGORY (-ACAT) Bit: The -ACCESS_CATEGORY bit is set if the command processor is matching access categories when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is an access category.

-RBF Bit: The -RBF bit is set if the command processor is matching RBF files when checking wildcard-laden names. It does not necessarily mean that the file system object specified in the current invocation is an RBF file. (RBF files are reserved for use by Prime.)

-VERIFY (-VFY) Bit: The -VERIFY bit is set if the command processor requires user verification of file system objects selected by wildcard-laden names. It does not necessarily mean that the user has verified the file system object specified in the current invocation, because verification is requested only if the user specifies a wildcard-laden name. Use the wildcard bit, described below, if you wish to determine whether the user was actually asked to verify the current invocation for the file system object; if both the -VERIFY bit and the wildcard bit are set (1), then verification was both requested and provided.

-BOTTOM_UP (-BOTUP) Bit: The -BOTTOM_UP bit is set (1) if the -BOTTOM_UP option (abbreviated -BOTUP) was specified on the command line, causing any treewalking to be performed at the lowest directory levels first. It does not necessarily mean that treewalking is being performed; see the treewalking bit, described below, for that information.

-WALK_FROM (-WLKFM) Value: The -WALK_FROM value is set to either the value specified following the -WALK_FROM option (abbreviated -WLKFM) on the command line or to the default value, which is 2. Level 1 is the contents of the directory itself; level 2 is the contents of the subdirectories, and so on. For example, in the treewalking specification DIR1>@@>FOO, level 1 is the DIR1 directory; if FOO exists in DIR1, it is found only if -WALK_FROM 1 is specified.

This value does not indicate whether treewalking is, in fact, being performed; see the treewalking bit, described below, for that information.

-WALK_TO (-WLKTO) Value: The -WALK_TO value is set to either the value specified following the -WALK_TO option (abbreviated -WLKTO) on the command line or the default value, which is 999. This value does not indicate whether treewalking is, in fact, being performed; see the treewalking bit, described below, for that information.

Iteration ( ) Bit: The iteration bit is set to '1'b if the command line used to invoke the program contained an iteration list (that is, contained parentheses). However, this bit is never set if the BIND subcommand NO_ITERATION (abbreviated NITR) was issued when the program was linked.

Wildcard @ + Bit: The wildcard bit is set to '1'b if the command line used to invoke the program contained a wildcard-laden entryname (that is, contained the @, +, or ^ character in the final element of a pathname or in a simple pathname). However, this bit is never set if the BIND subcommand NO_WILDCARD (abbreviated NWC) was issued when the program was linked.

Treewalk >@> >+> Bit: The treewalk bit is set to '1'b if the command line used to invoke the program contained a wildcard-laden directory name (that is, if it contained the @, +, or ^ character in a non-final element of a pathname). However, this bit is never set if the BIND subcommand NO_TREEWALK (abbreviated NTW) was issued when the program was linked.

Sample Program

The following sample PL/I-G program simply displays all of the information in the command processing information structure. While it is intended primarily to illustrate how to declare and use the command processing information structure in PL/I-G, it is also a useful program for experimenting with various combinations of command preprocessing features and BIND subcommands that enable, disable, or set parameters for command preprocessing features.

```
com_proc_info: proc(comline,code,cominfo);

dcl comline char(1024) var,          /* The command line. */
    code fixed bin(15),              /* Severity code. */
    1 cominfo,                       /* Command processing info. */
      2 comname char(32) var,        /* The command name. */
      2 version fixed bin(15),       /* Currently 0 or 1. */
      2 vcb_ptr ptr,                 /* CPL local variables. */
      2 preprocessing_info,          /* Command preprocessing info. */
        3 mod_after_date fixed bin(31),
                                     /* -MODIFIED_AFTER date. */
        3 mod_before_date fixed bin(31),
                                     /* -MODIFIED_BEFORE date. */
        3 bak_after_date fixed bin(31),
                                     /* -BACKEDUP_AFTER date. */
        3 bak_before_date fixed bin(31),
                                     /* -BACKEDUP_BEFORE date. */
        3 type_dir bit(1),           /* -DIR option specified. */
        3 type_segdir bit(1),        /* -SEGDIR option specified. */
        3 type_file bit(1),          /* -FILE option specified. */
        3 type_acat bit(1),          /* -ACAT option specified. */
        3 type_rbf bit(1),           /* -RBF option specified. */
        3 reserved_1 bit(11),        /* Reserved for future use. */
        3 verify_sw bit(1),          /* -VERIFY option specified. */
        3 botup_sw bit(1),           /* -BOTUP option specified. */
        3 reserved_2 bit(14),        /* Reserved for future use. */
        3 walk_from fixed bin(15),
                                     /* -WALK_FROM value. */
        3 walk_to fixed bin(15),     /* -WALK_TO value. */
        3 in_iteration bit(1),       /* In iteration sequence. */
        3 in_wildcard bit(1),        /* In wildcard sequence. */
        3 in_treewalk bit(1),        /* In treewalk sequence. */
        3 reserved_3 bit(13);        /* Reserved for future use. */

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl strings fixed bin(15),  /* Number of strings. */
    last_string char(80) var,  /* Last string. */
    line_to_show char(80) var;  /* Line waiting to be shown. */

dcl (tnoua,tnou) entry(char(80),fixed bin(15)),
    tovfd$ entry(fixed bin(15));

call tnoua('Command name is "',17);
call tnoua((comname),length(comname));
call tnoua('"',1);

if version=0 then
   do;  /* Version 0 means no more info. */
   call tnou('.',1);
   code=0;
   return;
   end;
```

```
if version=1 then;  /* Expected version number. */
else
   do;  /* New version, display it. */
   call tnoua(', version #',11);
   call tovfd$(version);
   end;  /* if version^=0 */

if vcb_ptr=null() then call tnou(', no CPL variables.',19);
else call tnou(', with CPL variables.',21);

call tnoua('Command line is "',17);
call tnoua((comline),length(comline));
call tnou('".',2);

strings=0;
last_string='';
line_to_show='Options: ';

if mod_after_date=0 then;
else call show_date('-MODIFIED_AFTER',mod_after_date);

if mod_before_date=0 then;
else call show_date('-MODIFIED_BEFORE',mod_before_date);

if bak_after_date=0 then;
else call show_date('-BACKEDUP_AFTER',bak_after_date);

if bak_before_date=0 then;
else call show_date('-BACKEDUP_BEFORE',bak_before_date);

if type_dir then call show_this('-DIR');
if type_segdir then call show_this('-SEGDIR');
if type_file then call show_this('-FILE');
if type_acat then call show_this('-ACAT');
if type_rbf then call show_this('-RBF');
if verify_sw then call show_this('-VERIFY');
if botup_sw then call show_this('-BOTUP');

if walk_from=2 then;  /* The default. */
else call show_value('-WALK_FROM',walk_from);

if walk_to=999 then;  /* The default. */
else call show_value('-WALK_TO',walk_to);

if in_iteration then call show_this('iteration');
if in_wildcard then call show_this('wildcard');
if in_treewalk then call show_this('treewalk');

/* Show last line if we have shown anything. */

if strings=0 then;
else
   if strings=1 then
```

```
      call tnou('Option: '||last_string,length(last_string)+8);
   else call show_this('');

code=0;
return;

show_date: proc(string,dtm);  /* Display option with date/time. */
dcl string char(32) var,
    dtm fixed bin(31);

dcl dow fixed bin(15),
    dtm_str char(21);

dcl cv$fda entry(bin(31),bin,char(21));

call cv$fda(dtm,dow,dtm_str);
call show_this(string||' '||trim(dtm_str,'11'b));

end;  /* show_date: proc */

show_value: proc(string,value);  /* Display option with integer. */

dcl string char(32) var,
    value fixed bin(15);

call show_this(string||' '||trim(char(value),'11'b));

end;  /* show_value: proc */

show_this: proc(string);  /* Display string in comma list. */

dcl string char(80) var;

dcl joiner char(6) var;

strings=strings+1;

if strings<=2 then joiner='';
else
   if string='' then
      if strings<=3 then joiner=' and ';
      else joiner=', and ';
   else joiner=', ';

if length(last_string)+length(line_to_show)+length(joiner)>79 then
   do;
   if strings<=3 then
      call tnou((line_to_show),length(line_to_show));
   else call tnou(line_to_show||',',length(line_to_show)+1);
   if string='' then line_to_show='and '||last_string;
   else line_to_show=last_string;
   end;
else
   line_to_show=line_to_show||joiner||last_string;
```

```
if string='' then call tnou((line_to_show),length(line_to_show));
else last_string=string;

end;  /* show_this: proc */

end;  /* com_proc_info: proc */
```

COMPLETE CALLING SEQUENCE

The complete calling sequence combines the command function calling sequence with the command processing information provided in the third argument of the calling sequence, as used in the detailed command calling sequence. In the command function calling sequence, described earlier, the third argument was ignored; in the detailed command calling sequence, as in the complete calling sequence, the third argument provides the program with information on the processing of the command that invoked the program.

Figure 3-7 illustrates the complete calling sequence, where EPF is the main entrypoint of the program EPF.

The first and second arguments are described in detail in the section entitled COMMAND CALLING SEQUENCE earlier in this chapter; the third argument is illustrated in Figure 3-6 and is described in the section entitled DETAILED COMMAND CALLING SEQUENCE earlier in this chapter; the fourth and fifth arguments are described in the section entitled COMMAND FUNCTION CALLING SEQUENCE. The remainder of this section explains why the complete calling sequence is useful and points out effects of combining a command and a command function in one program.

Why Use the Complete Calling Sequence?

A program that uses all five arguments in the complete calling sequence does so for one of several reasons:

- It is a command function that needs access to CPL variables local to the CPL program that called it.
- It is a command function that needs access to its own command name.
- It is a program that may be invoked as a command function or as a command, and when invoked as a command, it wishes to make use of command preprocessing information.
- It combines any of the above three reasons; for example, it might be a program that, when invoked as a command, does not need command processing information, but when invoked as a command function, needs the CPL Local Variables pointer.

Each of these uses of the complete calling sequence is examined in more detail in the next section.

Command Function Needing Local CPL Variables

When a command function needs access to the CPL variables local to the CPL program that invoked the command function, it uses the LV$GET and LV$SET subroutines to read and set the local CPL variables. An example of a command function that also sets local CPL variables is the [OPEN_FILE] function, described in the PRIMOS Commands Reference Guide and in the CPL User's Guide. Although not an EPF, this function could be written as an EPF as of Rev. 19.4, due to the program EPF interface described earlier in this chapter.

Command Function Needing Command Name

Rarely, a command function may need access to its command name, if it wishes to make a distinction (or to enforce an equivalence) between the name of the program as built during the BIND session that linked the program and the name of the program as invoked by the user. For example, when such a program issues messages, it may wish to use its invocation name, rather than its original name, so that its name may be easily changed without making error messages originating from the program difficult to track down.

Program Usable as a Command and as a Command Function

A program may need to be usable as both a command and as a command function. In addition, it may need access to command processing information when invoked as a command, as a command function, or as either one.

For example, a program may, when invoked as a command, wish to use command preprocessing information to generate useful output, depending upon whether its invocation included any wildcard, treewalking, or iteration specification. The same program, when invoked as a command function, does not need that information.

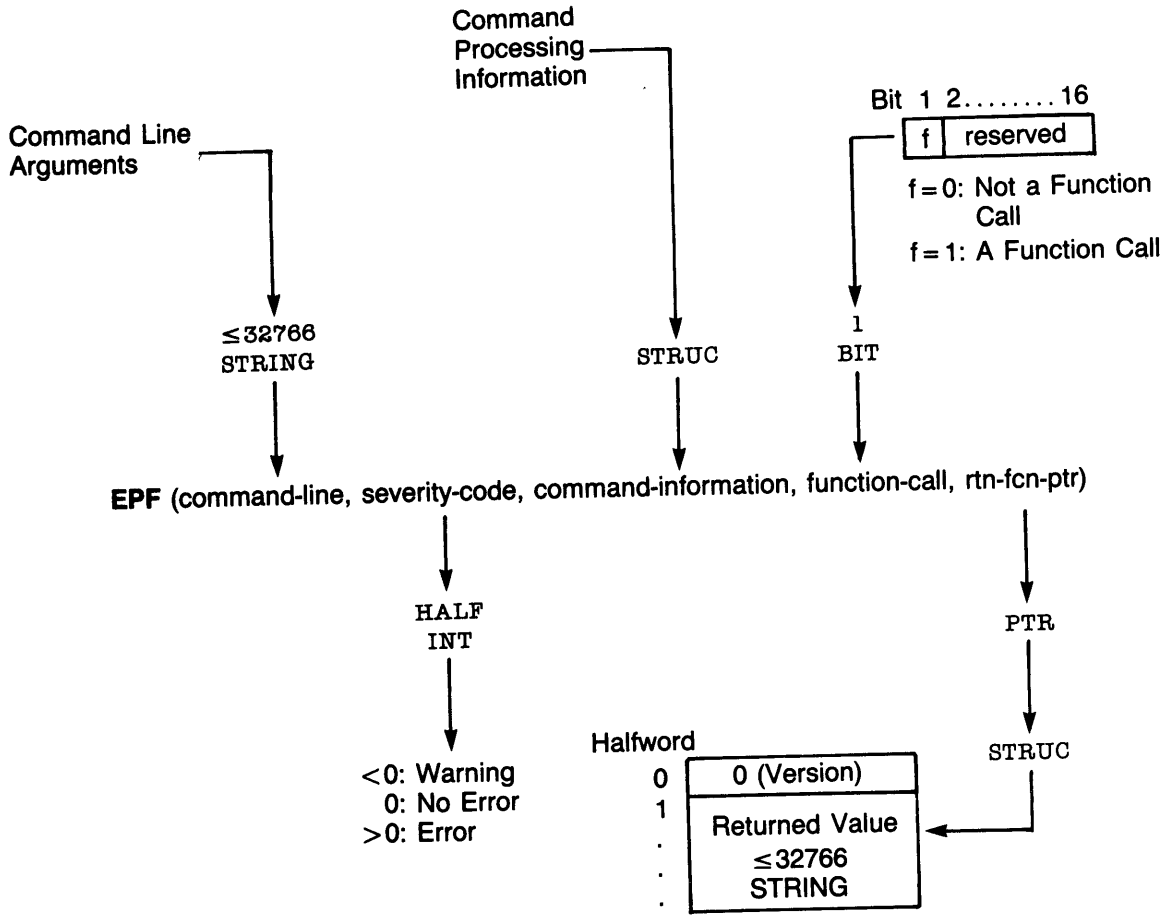
It is important to understand that the PRIMOS command processor does not perform any type of command iteration (including wildcarding, treewalking, and explicit iteration) when it is called upon to invoke a program as a command function.

Therefore, a program invoked as a command function should not expect to find any usable information in the command preprocessing information contained in halfword offsets 21-25 (25-31 octal) of the command processing information structure.

The PRIMOS command processor knows that a command is being invoked as a command function because its entrypoint, CP$, has a command-function bit as one of its input arguments. When this bit is set, CP$ does not perform any command iteration on the command line; instead, it passes the untouched command line directly through to the program EPF. (Other command preprocessing is performed as usual.)

However, a user-written command processor, other than CP$, may invoke a program EPF as a command function, providing useful information in halfword offsets 21-25 in the command processing information structure by passing it to EPF$INVK or EPF$RUN. If your program EPF is designed to be invoked only by such an application, it may use the command preprocessing iteration information even when invoked as a command function. This situation is expected to be quite rare.

Complete Calling Sequence



Complete Calling Sequence
Figure 3-7

4

Invoking Programs From Within Programs

A program or library may invoke another command, program, or function. PRIMOS provides three methods of invoking a program EPF, whether or not it is a function:

- Via the CP$ subroutine, which invokes the PRIMOS command processor
- Via the EPF$RUN subroutine, which invokes any program EPF
- Via the EPF$INVK subroutine, which invokes a program EPF that is already mapped to memory, allocated, and initialized

You may also use the CP$ subroutine to invoke a command, a program, a function, a CPL program, a CPL function, or a static-mode program.

This chapter describes how to use these subroutines to invoke commands, programs, and functions. This chapter also describes how to use the FRE$RA subroutine to free memory used to store the result of a command function. Finally, this chapter explains particular items that may be of interest when invoking other commands, programs, or functions.

COMMANDS, PROGRAMS, AND FUNCTIONS

There are several ways to categorize commands and programs under PRIMOS. From the point of view of the programmer who will be writing programs that invoke other programs or commands, the three most useful methods of categorizing commands and programs consider:

- Where the programming instructions for the command or program reside
- In which format the programming instructions for the command or program are stored
- Whether the command or program is invoked as a function (that is, whether it returns a value to its invoker)

In most cases, the PRIMOS command processor allows you to issue commands and run programs independent of their categorization. The interfaces described in this chapter, CP$, EPF$RUN, EPF$INVK, and FRE$RA pertain to different categories of commands and programs:

- CP$ can invoke any command or program, optionally as a function.
- EPF$RUN and EPF$INVK can invoke only a program EPF, optionally as a function.
- FRE$RA is used only at the completion of a function invocation, after the function has returned its value; it is used independently of the function's location or format.

Where the Programming Instructions Reside

The programming instructions for a command or program reside in one of the following locations:

- Internal to the PRIMOS Operating System
- On disk, in the CMDNC0 UFD
- On disk, but not in the CMDNC0 UFD

Commands are stored in the first two of these locations; programs are stored in the third. A command residing in the CMDNC0 UFD is just a program in a special place, and it may be run as a program; a program not residing in the CMDNC0 UFD may be made into a command simply by copying it into CMDNC0. Therefore, the distinction between commands and programs on disk is somewhat hazy; the terms "command" and "program" are often interchangeable, and are often used together in this guide. Some, but not necessarily all, commands and programs are supplied by Prime.

Internal to PRIMOS are internal commands. These are all Prime-supplied; Prime does not support the modification of PRIMOS by customers, such as adding new internal commands. Because internal commands reside in virtual memory rather than on disk, they are treated specially by the PRIMOS command processor. In fact, some internal commands have special privileges, such as the ability to access internal PRIMOS tables.

While user-written programs cannot always perform the same functions as internal PRIMOS commands, such programs can call the PRIMOS command processor to invoke internal PRIMOS commands.

A special internal PRIMOS command is the RESUME command, abbreviated R. The RESUME command is used to run a program. Therefore, the command processor treats a RESUME command as the invocation of a program rather than the invocation of an internal PRIMOS command. The special processing this involves is usually unimportant, except when handling errors and such.

Format of the Programming Instructions

The format of the programming instructions for a command or program is important to the PRIMOS command processor, because it determines how the command processor invokes the command or program. For commands and programs that reside on disk, there are three formats:

- Executable Program Format (EPF) Runfiles
- Command Procedure Language (CPL) Programs
- Static-mode Runfiles

(A fourth format, the SEG runfile, is not recognized by the PRIMOS command processor — it is recognized only by the SEG command, which itself is a static-mode runfile residing in the CMDNC0 UFD.)

Whether the PRIMOS command processor is called upon to execute a command in the CMDNC0 UFD or elsewhere on disk, it uses suffix searching to scan for the appropriate runfile. The suffixes .RUN, .SAVE, and .CPL are tried, in that order, and then a search with no suffix is tried. Based on the suffix that was in place when the runfile was found, the command processor infers the format of the runfile, as described in Chapter 1.

The most flexible format for programming instructions is the EPF, because a program written as a program EPF may be a function. In fact, it can determine whether it is being invoked as a function, and modify its actions accordingly. (The mechanism by which it does this is described in Chapter 3.)

In addition, a program EPF can modify CPL variables local to the CPL program that invoked it. Finally, a program EPF has the most control over selecting command processing features and determining which features are in use for a particular invocation.

The second most flexible format is the CPL program. A CPL program can be written either as a program or as a function. However, CPL programs cannot automatically determine whether they are being invoked as functions; but they can accept a command line option supplied by the invoker to indicate which type of invocation is taking place. Otherwise, a CPL program must assume either that it will always be invoked as a program or that it will always be invoked as a function.

CPL programs can also choose how they will handle wildcards, as wildcards are not processed for CPL programs.

The least flexible format is the static-mode program. A static-mode program cannot be written as a function. If the name of a static-mode program begins with NX$ or NW$, this disables various combinations of command processing features. This naming scheme represents the only control that static-mode programs have over command processing features; and it requires users to enter the NX$ or NW$ prefix when entering the program name.

For commands internal to PRIMOS, there is only one format, and that is the format of a subroutine, or procedure, that accepts a standardized calling sequence as its arguments.

Functions

A function returns a value to the invoker of the function. This value typically replaces the invocation of the function (in a CPL program command line, for example).

Almost all Prime-supplied functions are commands, either internal to PRIMOS or residing in CMDNC0. Functions that are commands are often called command functions. Prior to Rev. 19.4, users could write functions only in CPL; as of Rev. 19.4, they may write functions as program EPFs. The term program function can be used to refer to a function not supplied by Prime; however, this distinction is not usually important for readers of this guide. Therefore, the terms function and command function are used generically to refer to any command or program that returns a function value when invoked as a function.

The difference between a program that is a function and one that is not is whether the program is designed to operate as a function and whether the invoker of the program is invoking it as a function.

For example, the ABBREV -STATUS command, when used as a command, does not operate as a function — it displays the pathname of the user's abbreviation file, and the number of abbreviations defined in the file.

```
OK, ABBREV -STATUS
Abbreviation file: UNGER>LOGIN.ABBREVS
Abbreviations: 183

OK,
```

When used as a function, however, ABBREV -STATUS modifies its behavior so that it displays nothing to the terminal. Instead, it returns the pathname of the user's abbreviation file as the value of its invocation:

```
OK, TYPE Your abbreviation file is: [ABBREV -STATUS]
Your abbreviation file is: UNGER>LOGIN.ABBREVS
OK,
```

The displayed output came not from the ABBREV -STATUS invocation, but from the TYPE command.

The ABBREV -STATUS command is an example of a command that operates as either a command or as a function, depending on how it is used. Typically, however, a command or program always operates as one or the other. For example, another internal command, RDY, operates only as a command — when invoked as a function, it still behaves as a command and returns no value:

```
OK, TYPE Value of RDY command is: [RDY]
OK 14:33:39  243.024  11.354
Value of RDY command is:
OK,
```

The first line of displayed output came from the invocation of the RDY command. The second line of output came from the invocation of the TYPE command, which included a function invocation of RDY that returned no result because RDY is not a function.

Conversely, a command or program may be constructed to run only as a function. For example, when invoked as a command, the internal command SUBSTR detects that it has not been invoked as a function, displays an error message, and returns a positive severity code (producing the ER! prompt):

```
OK, SUBSTR TEST 2 2
May only be invoked as a command function. (SUBSTR)
ER!
```

DECIDING WHICH INTERFACE TO USE

To write a program, library, or subroutine that invokes another command, program, or function, you must first decide which interface to use:

- CP$
- EPF$RUN
- EPF$INVK
- FRE$RA

You make your decision based on what kind of program you wish to invoke, and whether you wish to use command preprocessing features such as variable expansion, wildcarding, and name generation.

- Use CP$ to invoke a PRIMOS command or a program, or to include command preprocessing features.
- Use EPF$RUN to invoke a program EPF.
- Use EPF$INVK to invoke a program EPF with more control over how and when the EPF is set up.
- Use FRE$RA only if you invoke a function and accept a returned text string.

Typically, you choose only one of the CP$, EPF$RUN, and EPF$INVK subroutines; these allow your program to invoke either a program or a function. After calling a function, your program makes use of the returned text string. Your program then calls the FRE$RA subroutine to free the memory used to store the returned text string, allowing the memory to be reused.

When to Use CP$

You use the CP$ subroutine to invoke:

- Internal PRIMOS commands, such as ASSIGN
- External CPL programs
- External EPFs
- External static-mode programs

Except for external static-mode programs, any of the above may be invoked as functions.

Calling CP$ invokes the PRIMOS command processor, STD$CP. This same command processor is invoked when the user enters a response to the OK, prompt issued by PRIMOS.

User-defined abbreviations are not expanded by CP$. Therefore, you can reliably use CP$ in your program without concerning yourself with user-defined abbreviations that might change the meaning of your command lines. For example, calling CP$ to invoke the ASSIGN MT0 command always invokes that command, even if the user has defined ASSIGN or MT0 as an abbreviation via the PRIMOS abbreviation facility.

The PRIMOS command processor, invoked via CP$, determines what command is being executed as follows:

1. The first token of the command line is parsed. This is the name of the command being invoked. For example, consider the command line:

   ```
   COPY FRED>MEMO.12/31/84 *>MEMOS>MEMO.118
   ```

   Here, the name of the command is COPY.

2. The command name is checked against the list of internal PRIMOS commands. One important internal PRIMOS command is RESUME; if the command is RESUME, the program specified by the pathname following the RESUME command is invoked.

   If the command name is not RESUME, and is found in the list of internal PRIMOS commands, the appropriate command line preprocessing (such as wildcarding) is performed, and the internal PRIMOS subroutine that corresponds to the command name is invoked. The command processor returns to the caller when the internal PRIMOS subroutine has finished.

3. If the command name is not in the list of internal PRIMOS commands, the command processor searches the CMDNC0 directory for a program with the same name as the command. If found, the program is executed as if it had been RESUMEd.

When executing a program, the command processor first performs the appropriate command preprocessing (such as wildcarding), depending upon the program type. If the program is an EPF, the command preprocessing is determined by information that is placed within the EPF itself when the EPF is built using BIND subcommands. For information on BIND subcommands that describe the command preprocessing environment for an EPF, see Chapter 2. See the PRIMOS Commands Reference Guide for information on command preprocessing for static-mode and CPL programs.

Although command programs reside only in the CMDNCO directory, CP$ can be used to invoke programs residing anywhere on disk, by invoking the internal command RESUME via CP$. For example, to invoke the program ACCOUNTS_PAYABLE in the current directory, call CP$ with the following command line:

```
RESUME ACCOUNTS_PAYABLE
```

When to Use EPF$RUN

You use EPF$RUN to invoke a program EPF. As with CP$, you pass the command line to the target program, but no command preprocessing is performed on the command line. Therefore, use EPF$RUN when you do not want any changes to be made to the command line being passed.

EPF$RUN handles all of the tasks needed to execute a program EPF, including mapping the EPF to memory, allocating the linkage area, initializing the linkage area, and optionally removing the EPF from memory when the invocation has been completed.

When to Use EPF$INVK

You use EPF$RUN to invoke a program EPF that has already been mapped to memory, allocated, and initialized. As with CP$, you pass the command line to the target program, but no command preprocessing is performed on the command line. Therefore, use EPF$INVK when you do not want any changes to be made to the command line being passed.

The advantage of using EPF$INVK over EPF$RUN is that you have more control over the phases of EPF execution. However, you must call several other subroutines, described in this chapter, to map the EPF to memory, to allocate the linkage area, to initialize the linkage area, and to remove the EPF from memory after invocation.

When to Use FRE$RA

You use the FRE$RA subroutine after using CP$, EPF$RUN, or EPF$INVK to invoke a function only if the returned function pointer is not a null pointer (segment number 7777). Your program should call FRE$RA sometime after it finishes using the returned function value; this may be after it makes its own copy of the value, or after it finishes analyzing the value. If you have used EPF$INVK to invoke the function, it is not important whether your program calls FRE$RA before or after calling EPF$DEL to remove the EPF.

THE CP$ SUBROUTINE

There are two ways of using CP$:

- Invoking commands or programs
- Invoking functions

The calling sequence for CP$ has six arguments. When not invoking a function, you may wish to pass only three arguments; the remaining three arguments are assigned default values before being passed to the PRIMOS command processor, STD$CP.

Figure 4-1 illustrates the calling sequence for CP$. The next two sections describe how to use CP$ to invoke a command, program, or function.

Using CP$ to Invoke a Command or Program

To use the CP$ subroutine to invoke an internal PRIMOS command or a program, rather than a function, you typically need to supply only the first three arguments — command-line, code, and severity-code — of the calling sequence illustrated in Figure 4-1. If you wish to pass a pointer to local CPL variables, then you must supply five or six arguments in the calling sequence to include the cpl-local-vars-ptr argument.

Before calling CP$, your program should initialize the severity-code argument to 0, in case it is not set by the command or program being invoked.

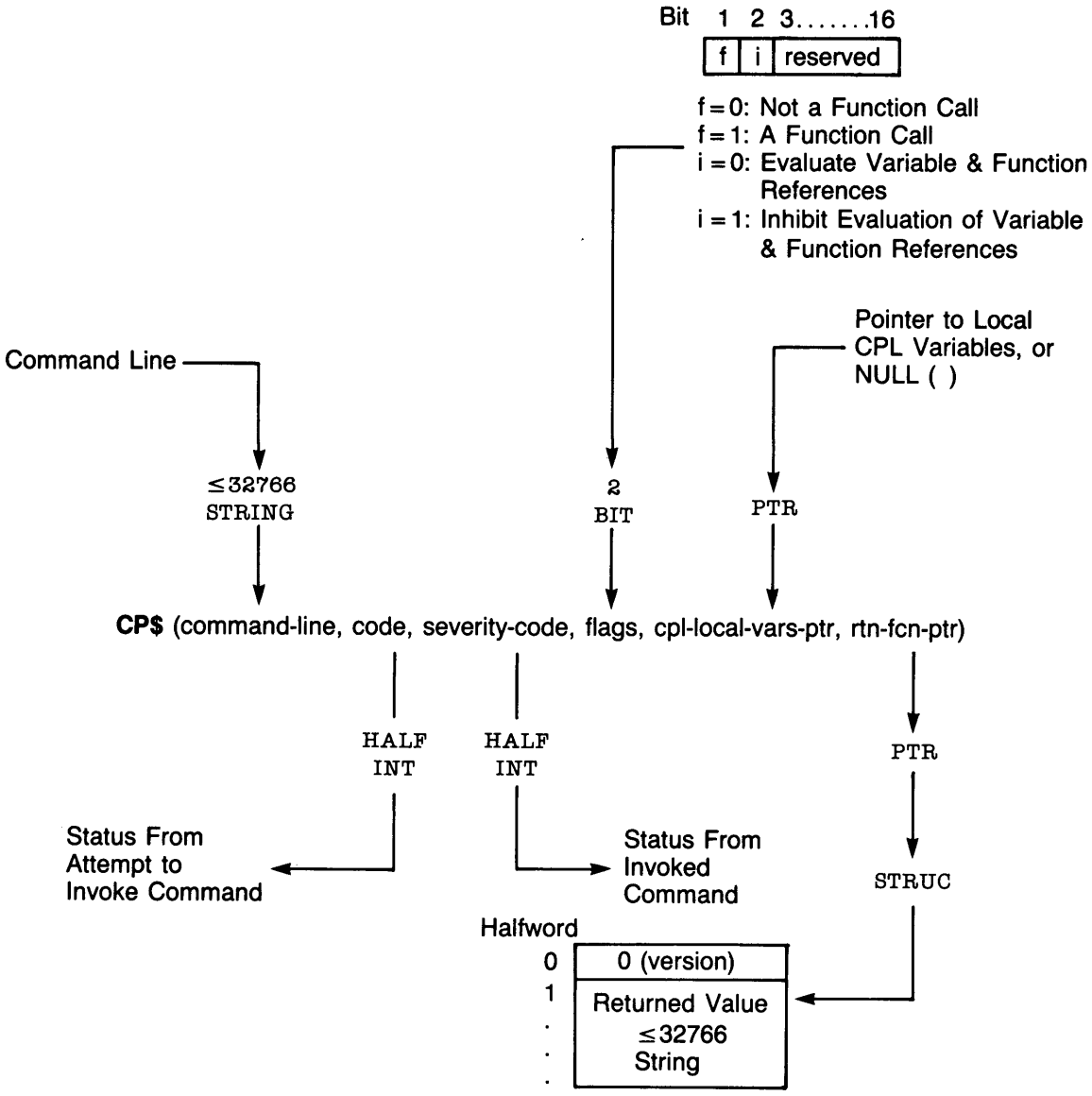
When your program calls CP$, the command processor attempts to execute the command passed in command-line. If it fails to begin execution, a standard PRIMOS error code is returned in code. If it succeeds in executing the command, 0 is returned in code, and the status of the command itself is returned in severity-code.

Ultimately, when the program you invoke via a call to CP$ is a program EPF, the severity-code argument to CP$ corresponds to and is set from the severity-code argument in the calling sequence for a program EPF, described in Chapter 3; CPL programs set this value by issuing a &RETURN directive, and static-mode programs set this value by calling the SETRC$ subroutine.

Note

The returned value of severity-code is undefined if the returned value of code is nonzero.

Invoke a Command, Program, or Function



Calling Sequence of CP$
Figure 4-1

The Command Line: In command-line, simply pass the command line that you would type as a user invoking the command. The PRIMOS Commands Reference Guide contains information on command formats. For example, to assign a magnetic tape drive for a running program, you might have your program call CP$ with the command line:

```
ASSIGN MT0 -WAIT
```

The RESUME command is a special case, because it is an internal command that runs an external program. Use the RESUME command to invoke a program via CP$. For example, to run a program, you might have your program call CP$ with the command line:

```
RESUME MYPROG MEMO.03/08/05
```

Unless you place a tilde (~) in front of the command line, CP$ performs certain kinds of command preprocessing on command-line before actually invoking the internal command (although it never modifies command-line itself). First, if the command line contains one or more unquoted command separator characters (;), CP$ splits up the command line into several separately handled command lines.

Then, unless inhibited by the second bit of flags, CP$ resolves command function references and variable references. Subsequent command preprocessing depends on the command or program being invoked; for example, ATTACH does not accept wildcards, but LIST_QUOTA does. See the PRIMOS Commands Reference Guide for information on command preprocessing support by Prime commands; use the LIST_EPF -COMMAND_PROCESSING command to determine what kind of command preprocessing is performed for a particular program EPF being invoked.

Note

Placing a tilde (~) in front of the command line as passed to CP$ has the effect of preventing all forms of command preprocessing. Therefore, calling CP$ with the command line

```
~SET_VAR .FOO %OPTION% is an option; [SET_1] is a function.
```

causes the global variable .FOO to be set to exactly the string shown. Without the tilde, the variable %OPTION% and command function reference [SET_1] would be evaluated, and the results would be substituted in the command line (assuming the variable and function references succeeded). In addition, the semicolon after "option" would be treated as a command separator.

The Error Code: The code argument, returned by CP$, indicates the degree of success encountered by the command processor's attempt to execute the command. For example, if the command is not found, the error code e$fntf (Not found) is returned in code.

Any nonzero value returned in code indicates that all other output arguments have undefined values, because they all depend upon the successful invocation of the command.

See the section entitled Error Codes From CP$, later in this chapter, for a partial list of error codes; see Volume 0 of this series for a complete list of PRIMOS error codes.

The Severity Code: The severity-code argument, returned by the invoked command via the command processor and CP$, indicates the degree of success reported by the invoked command. For example, if you invoke the ATTACH command to attach to a nonexistent subdirectory, the error code e$fntf (Not found) is returned in severity-code.

Note

The RESUME command is handled by the command processor in a special way. The target of the RESUME command is the program to be invoked. If the target program is not found, the error code is returned in code, not in severity-code as for other commands (such as ATTACH, COPY, and so on). This allows the calling program to distinguish between a missing program and a program that cannot find the target specified on its command line.

The Function-Call Bit: The first bit of the flags argument specifies whether the call to CP$ is to invoke a function (such as GVPATH or a user-written function) or not. If flags is not supplied in the calling sequence, the function-call bit defaults to 0, meaning that a function invocation is not being made. If flags is supplied, set this bit to 0 to indicate that you are invoking a command or program rather than a function. (The use of CP$ to invoke a function is described in the next section.)

The Inhibit-Evaluation Bit: The second bit of the flags argument specifies whether command function references and variable references in the command line are to be evaluated. If flags is not supplied in the calling sequence, the inhibit-evaluation bit defaults to 0, meaning that such references are to be evaluated. If flags is supplied, set this bit to 0 if you wish such references to be evaluated, or set this bit to 1 if you wish such references to be passed to the target program instead of being evaluated.

The CPL Local Variables Pointer: The cpl-local-vars-ptr argument provides the necessary "toehold" for the target command or program to set CPL variables local to the procedure that invoked your program. Typically, you either do not supply this argument or you supply the null pointer (NULL(), which is segment 7777 offset 0). If you do not pass this argument, CP$ substitutes the null pointer when calling the PRIMOS command processor, STD$CP.

If your program may be invoked by a CPL program, and if it is using CP$ to invoke a program that may need to set one or more CPL variables local to the invoking CPL program, then your program should pass, in cpl-local-vars-ptr, the corresponding pointer that was passed to its main entrypoint in the command-information structure of the program EPF calling sequence. (See Chapter 3 for more information on the command-information structure.)

The Returned Function Value Pointer: The rtn-fcn-ptr argument is not used when invoking a command or program. It is used only when invoking a function, that is, when bit 1 of the flags argument is set to 1, as described in the next section.

Using CP$ to Invoke a Function

The CP$ subroutine may be used to invoke a command function that is either an internal PRIMOS command function, such as DATE and GVPATH, or a user-written command function, written in CPL or as a program EPF. Whether the command function being invoked is a Prime-supplied command function or a user-written command function, your program calls CP$ in the same way.

To use the CP$ subroutine to invoke a function, have your program pass all six arguments to CP$ as illustrated in Figure 4-1 earlier in this chapter.

Before calling CP$, your program should initialize the severity-code argument to 0 and the rtn-fcn-ptr to the null pointer (NULL() in PL/I-G), in case these arguments are not set by the function being invoked.

When your program calls CP$, the command processor attempts to execute the command passed in command-line. If it fails to begin execution, a standard PRIMOS error code is returned in code. If it succeeds in executing the command, 0 is returned in code, the status of the command itself is returned in severity-code, and a pointer to the returned text string structure is returned in rtn-fcn-ptr.

Ultimately, when the program you invoke via a call to CP$ is a program EPF, the rtn-fcn-ptr argument to CP$ corresponds to the rtn-fcn-ptr argument in the calling sequence for a program EPF, described in Chapter 3; CPL programs set this value by issuing a &RESULT directive.

Notes

1. The returned values of severity-code and rtn-fcn-ptr are undefined if the returned value of code is nonzero.

2. When invoking a command function, no wildcarding, iteration, or treewalking is performed. In addition, the command separator character, the semicolon (;) is not honored; is treated like any other character.

The Command Line: In command-line, use the RESUME command, or the command name itself, just as you would when invoking a command or program. Do not enclose the command line in square brackets ([ ]) as you would in a CPL program.

For example, to determine the user's abbreviation file, call CP$ with the command line:

```
ABBREV -STATUS
```

The pathname of the abbreviation file, -OFF, or both, is returned in the structure pointed to by rtn-fcn-ptr, as described below.

To invoke a user-written command function, you might have your program call CP$ with the following command line:

```
RESUME PROGRAMS>GET_RECORD 1154 -DATABASE PAYROLL
```

Again, the information is returned in a structure pointed to by rtn-fcn-ptr.

Unless you place a tilde in front of the command line or set the second bit of flags to 1, CP$ resolves (nested) command function references and variable references.

The Error Code: The code argument, returned by CP$, has the same meaning for function invocation as for command or program invocation, described earlier in this chapter.

The Severity Code: The severity-code argument, returned by the invoked function via the command processor and CP$, has the same meaning for function invocation as for command or program invocation, described earlier in this chapter.

The Function-Call Bit: The first bit of the flags argument specifies whether the call to CP$ is to invoke a function (such as GVPATH or a

user-written function) or not. Set this bit to 1 to indicate a function invocation.

The Inhibit-Evaluation Bit: The second bit of the flags argument has the same meaning for function invocation as for command or program invocation, as described earlier in this chapter.

The CPL Local Variables Pointer: The cpl-local-vars-ptr argument has the same meaning for function invocation as for command or program invocation, as described earlier in this chapter.

The Returned Function Value Pointer: The rtn-fcn-ptr argument contains a pointer to the returned function value when CP$ returns, or the null pointer if no function value has been returned. Actually, rtn-fcn-ptr points to a structure that contains the returned value, as illustrated in Figure 4-1.

Note

If the invoked command did not return a value, then rtn-fcn-ptr may not have been modified. Therefore, set it to the null pointer before calling CP$, and check it after CP$ returns to make certain that a result has been returned.

In PL/I-G, the declaration of the returned function value structure is:

```
dcl 1 rtn_function_structure based(rtn-fcn-ptr),
      2 version fixed bin(15),
      2 text_string char(32766) var;
```

If version does not contain 0, do not attempt to use text_string, because a nonzero version indicates a new version of the returned structure. However, version should contain 0, and text_string should contain the returned text string.

After using the returned text string, your program should free the returned text string structure to the pool of available memory. Use the FRE$RA subroutine to do this. FRE$RA is described later in this chapter.

If your program is written in FORTRAN, access to the returned function value is difficult. Here is a programming discipline that allows a FORTRAN program to copy the returned function value, pointed to by an INTEGER*4 pointer variable named RFNPTR, into an INTEGER*2 array of characters named RTNFCN and a length variable named RTNLEN. The maximum number of characters that can be held by RTNFCN is set in a parameter named RTNMAX.

```
      INTEGER*2 GCHAR,IXS,IXD,RTNFCN(512),RTNLEN,RTNMAX
      INTEGER*4 RFCPTR
C
      PARAMETER RTNMAX=1024
C ...
C ... CALL CP$ HERE, check error code
C ...
C
C Check if the returned pointer is the null pointer.
C
      IF (AND(RFCPTR,:177600000).NE.:17760000) GO TO 98710
C
C Null pointer, assume zero-length result.
C
      RTNLEN=0
      GO TO 98800  /* Do not call FRE$RA with a null pointer!
C
C Have a pointer, see if version 0.
C
98710 IXS=0  /* Source string index.
      IF (GCHAR(RFNPTR,IXS)+GCHAR(RFNPTR,IXS).EQ.0) GO TO 98720
C
C Not version 0, unknown version, assume null value.
C
      RTNLEN=0
      GO TO 98790  /* Do call FRE$RA to deallocate the structure.
C
C Get length of returned function value in RTNLEN.
C
98720 RTNLEN=LS(GCHAR(RFNPTR,IXS),8)+GCHAR(RFNPTR,IXS)
C
C Now, IXS should be 4 which is the beginning of the value itself.
C Copy the value into RTNFCN until the end of the source or the end
C of the destination is reached.
C
      IF (RTNLEN.EQ.0) GO TO 98790  /* Null value!
C
      IXD=0  /* Destination string index.
C
C Loop until string copied.
C
98730 CALL SCHAR(LOC(RTNFCN),IXD,GCHAR(RFNPTR,IXS))
      IF (IXS.LT.RTNLEN.AND.IXD.LT.RTNMAX) GO TO 98730
C
C Now free the structure.
C
98790 CALL FRE$RA(RFNPTR)
C
C Done!
C
98800 CONTINUE
```

Error Codes From CP$

An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT value. Symbols are provided to allow PL/I-G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings follow. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by CP$, but are not listed.

Note

When you use CP$ to invoke a program EPF, either via the RESUME command or by specifying a program EPF in CMDNCO, an error code returned by the EPF$RUN subroutine is returned by CP$. Therefore, consult the list of error codes returned by EPF$RUN, later in this chapter, for information on additional error codes returnable by CP$.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$EOF	1	End of file. Typically, this error indicates an attempt to invoke a text file (such as a CPL file) as a static-mode program. Alternatively, this error indicates a file that has been truncated by FIX_DISK during system maintenance procedures. In the latter case, you must replace the program with a backup copy.
E$FIUS	3	File in use. Indicates an attempt to run a program that is open for writing.
E$NRIT	10	Insufficient access rights. You do not have access to the program.
E$DIRE	14	Operation illegal on directory. Typically, this error indicates an attempt to invoke a segment directory, such as a .SEG file, with the RESUME command. Alternatively, this error indicates an attempt to invoke a file directory.

E$FNTF	15	Not found. If the command is the RESUME command, the target program could not be found. Otherwise, the command is not an internal command, and a program with the same name could not be found in CMDNC0.
E$BNAM	17	Illegal name. The RESUME command specifies a filename not conforming to filename syntax rules.
E$ITRE	57	Illegal treename. The RESUME command specifies a pathname not conforming to pathname syntax rules.
E$CMND	68	Bad command format. The command name, the first token on the command line, is more than 32 characters long or does not conform to filename syntax rules.
E$BARG	71	Invalid argument to command. The RESUME command is not followed by a program name.
E$NDAM	109	Not a DAM file. The target program is a .RUN file, indicating an EPF, but is not a DAM file. The fault is in the installation of the program being invoked.
E$BVER	158	Incorrect version number. Typically, this error means that the command function invoked by the call to CP$ returned a structure containing an invalid version number. Alternatively, this error means that the target EPF contains an invalid version number. In both cases, the fault is in the command function, not the calling program. The command function is an EPF, because a CPL program should never cause this error. If the command function is in fact a CPL program, contact your Customer Support Center.
E$NINF	159	No information. You do not have access to the program.

THE EPF$RUN SUBROUTINE

The EPF$RUN subroutine is used in the following manner:

1. The calling program opens the program EPF file to be invoked.
2. The calling program calls EPF$RUN, passing the file unit number of the opened program EPF file.

3. The calling program closes the program EPF.

4. After the EPF$RUN subroutine completes, the calling program checks the returned error code to determine whether the program EPF was successfully invoked by EPF$RUN.

5. If the error code from EPF$RUN is 0, the calling program uses the information returned by EPF$RUN to determine whether the program EPF completed successfully or unsuccessfully, and — if the program EPF was invoked as a command function -- to access the returned text string.

6. If the error code from EPF$RUN is 0, and the calling program invoked the program EPF as a command function, the calling program uses the FRE$RA subroutine to return the memory used to store the returned text string to the free memory pool.

These steps are described in detail below. They are followed by a list of error codes that may be returned by EPF$RUN.

Step 1: Open the EPF File

Your program must first open the target program EPF file for VMFA-read before calling EPF$RUN. VMFA (Virtual Memory File Access) provides efficient data retrieval from disk storage by mapping disk records into memory via the virtual memory mechanism. PRIMOS implements a limited form of VMFA called read-only VMFA, and supports this mechanism for use only by the EPF mechanism.

To open the target program EPF file for VMFA-read, use the k$vmr key when you invoke the SRCH$$, TSRC$$, or SRSFX$ subroutines. For example, a PL/I-G program might use the following call:

```
call srsfx$(k$vmr+k$getu,'MY_EPF',unit,type,1,'.RUN',basename,i,
     code);
```

A FORTRAN program might use the following statement:

```
CALL SRCH$$(K$VMR+K$GETU,'MY_EPF.RUN',10,UNIT,TYPE,CODE)
```

Typically, you add k$getu to the k$vmr key, to specify that a free file unit is to be found by PRIMOS. If you do, the file unit number used is returned in unit. If you do not add k$getu, you must pass a valid file unit number in unit.

If code is 0 when SRSFX$, TSRC$$, or SRCH$$ returns, the file is open on the indicated file unit. Otherwise, the file is not open, and code contains an error code indicating the problem. If an error occurred, EPF$RUN cannot be called to invoke the EPF, because it is not open.

See the Subroutines Reference Guide for details on the SRSFX$, TSRC$$, and SRCH$$ subroutines.

Step 2: Invoke EPF$RUN

After your program has opened the target program EPF file, it calls EPF$RUN. Figure 4-2 illustrates the calling sequence for the EPF$RUN subroutine.

Although the calling sequence contains eight arguments, there are two cases in which only the first three arguments need be passed. These cases are:

- When the k$restore_only value for key is used, in which case the target EPF is not actually invoked
- When the main entrypoint of the target EPF is known to accept no arguments

The other five arguments are not used by EPF$RUN or by EPF$INVK (which EPF$RUN calls to invoke the EPF). They are simply passed to the main entrypoint of the program EPF, corresponding to the five arguments in the complete calling sequence of a program EPF, described in Chapter 3.

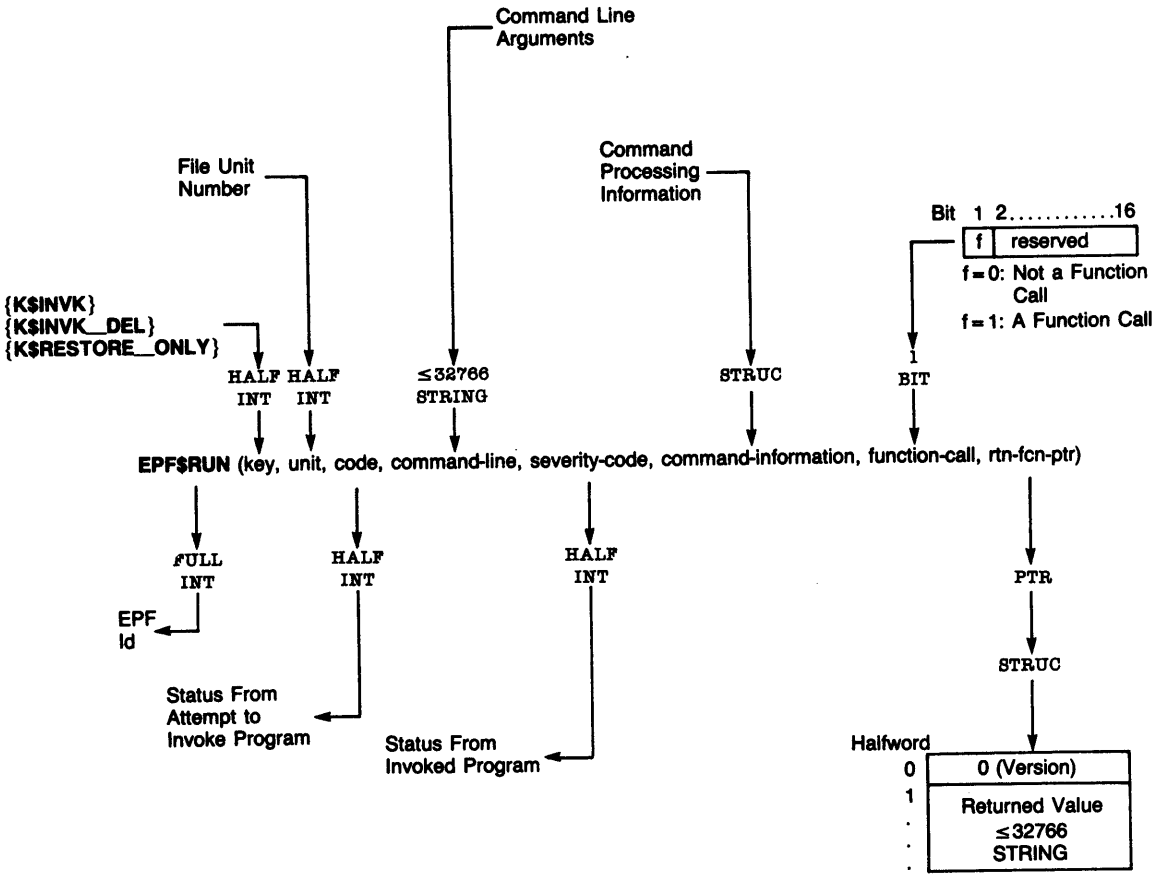
The arguments for the EPF$RUN subroutine are described below.

The Key: For key, specify k$invk, k$invk_del, or k$restore_only. Both k$invk and k$invk_del cause the target EPF to be invoked; however, k$invk causes the program EPF to be left in the EPF cache after it completes, whereas k$invk_del causes the program EPF to be removed from the EPF cache after it completes.

The k$restore_only key causes all activities up to, but not including, the invocation of the program EPF to be performed; use the EPF$INVK and EPF$DEL subroutines, described later in this chapter, to complete the process of executing the program EPF.

The EPF cache is a mechanism in PRIMOS to optimize frequent reuse of EPFs. Therefore, use the k$invk key if the target program EPF may be invoked more than once by the program or user. Use the k$invk_del key if you are certain that the invocation of the target program EPF by the calling program will be the last such invocation by that user for some time.

Run a Program EPF



Calling Sequence of EPF$RUN
Figure 4-2

The File Unit: Pass the file unit on which the target program EPF is open for VMFA-read (from Step 1) in unit.

The Error Code: When EPF$RUN returns, the value in code indicates the success or failure of the operation. If code is 0, the target program EPF was successfully invoked, although it may not have completed successfully.

If code is not zero, an error occurred while trying to invoke the EPF. In this case, your program should display an error message (using the ERRPR$ subroutine) and perhaps log the error; however, your program should not make use of any other information returned by EPF$RUN, such as severity-code or rtn-fcn-ptr, because these variables are assigned only as a result of successful invocation of the EPF.

See the section entitled Error Codes From EPF$RUN, later in this chapter, for a partial list of error codes.

The Command Line: Pass the command line containing the command arguments for the target program EPF in command-line; if there are no arguments, pass the null string.

Note

Do not include the RESUME command or the program name in the command-line argument. Otherwise, the target program EPF treats the RESUME command as the first token in the command line, and the pathname of the program as the second token, rather than treating the information following RESUME program-name as the command line.

The Severity Code: When the EPF$RUN subroutine returns, if code is 0, severity-code contains the severity code of the invoked EPF. The interpretation of severity-code is strictly dependent on the program EPF itself; however, it is typically set and interpreted as follows:

Value	Meaning
0	Program completed successfully
< 0	Successful completion, defined operation not performed (warning)
> 0	Program did not complete successfully (error)

Note

Because severity-code may not be set by the target program EPF, preset it to 0 before calling EPF$RUN, so that the default value indicates successful completion. This is particularly important when invoking a program that does not use its command line to receive information, and hence may have a main entrypoint that does not accept any arguments.

The Command Information: There are currently two versions of the command information structure that your program can pass. Both of these versions are illustrated in Chapter 3. Typically, you can pass a version 0 structure, which contains only the command name and the version number. If your program must pass a pointer to local CPL variables, or if your program performs command preprocessing such as wildcards, it must pass a version 1 structure.

In PL/I-G, version 0 of the command-information structure is declared as follows:

```
dcl 1 command_state static,
      2 command_name char(32) var init(''),
      2 version fixed bin(15) init(0);
```

Version 1 of the command-information structure is declared as follows:

```
dcl 1 command_state static,
      2 command_name char(32) var init(''),
      2 version fixed bin(15) init(1),
      2 cpl_local_vars_ptr ptr init(null()),
      2 cp_iter_info,  /* Command iteration info. */
        3 mod_after_date fixed bin(31) init(0),
        3 mod_before_date fixed bin(31) init(0),
        3 bk_after_date fixed bin(31) init(0),
        3 bk_before_date fixed bin(31) init(0),
        3 type_dir bit(1) init('1'b),
        3 type_segdir bit(1) init('1'b),
        3 type_file bit(1) init('1'b),
        3 type_acat bit(1) init('1'b),
        3 type_rbf bit(1) init('0'b),
        3 mbz1 bit(11) init('00000000000'b),
        3 verify_sw bit(1) init('0'b),
        3 botup_sw bit(1) init('0'b),
        3 mbz2 bit(14) init('00000000000000'b),
        3 walk_from fixed bin(15) init(2),
        3 walk_to fixed bin(15) init(999),
        3 in_iteration bit(1) init('0'b),
        3 in_wildcard bit(1) init('0'b),
        3 in_treewalk bit(1) init('0'b),
        3 mbz3 bit(13) init('0000000000000'b);
```

Before calling EPF$RUN, set command_name to the name of the target program EPF you are invoking (32 characters maximum). If you know the name of the program while writing the program, you may place the name in the INITIAL attribute for the declaration of command_name. If, in Step 1, your program called SRSFX$, then store the basename variable, returned by SRSFX$, in command_name. command_name should not contain the .RUN suffix of the program. The degree of flexibility you have in setting command_name depends solely upon the program EPF you are invoking; therefore, consult the specification for the appropriate program.

The INITIAL attributes used in the declaration above indicate the default settings used by PRIMOS. If your program is performing wildcard selection, matching, treewalking, and so on, you may wish to have your program modify cp_iter_info appropriately.

If the program being invoked references CPL variables local to the CPL program that invoked it (and therefore the CPL program that invoked your program EPF), store the pointer passed to the main entrypoint of your program EPF (in the command-information structure argument) into cpl_local_vars_ptr before calling EPF$RUN. See Chapter 3 for more information on the command-information structure passed to program EPFs.

Function Call: The function-call bit indicates to the target EPF whether the EPF should return a function value. If you do not intend to use the target program EPF as a command function, set this bit to 0. If you do intend to use the target program EPF as a command function, set this bit to 1.

The Returned Function Value Pointer: The rtn-fcn-ptr variable has the same meaning for EPF$RUN as it does for CP$, as described earlier in this chapter.

The EPF Id: The returned value of EPF$RUN, when invoked as a function that returns a FULL INT value, is an internal PRIMOS identifier of the EPF that is valid only if code is 0 and your program did not supply a key value of k$invk_del. You may use this identifier in subsequent calls to EPFCPF, EPFINVK, and EPF$DEL, which are described below.

You do not need to declare EPF$RUN as a function if you do not intend to use the returned EPF identifier.

Step 3: Close the EPF File

After EPF$RUN returns, close the file unit on which the target program EPF is open by calling SRCH$$. For example:

```
call clo$fu(unit,i);  /* Don't overwrite CODE! */
```

Note

It is not necessary to repeatedly open and close a program EPF file when repeated invocations of the EPF are to be performed. The program EPF file can be opened once, invoked several times via EPF$RUN, and then closed once.

Step 4: Check the EPF$RUN Error Code

After closing the EPF file, check the returned code value. If code is 0, proceed to step 4. Otherwise, code contains a standard PRIMOS error code; use ERRPR$ or ERTXT$ to report the error to the user or to log the error. A listing of possible error codes that may be returned by EPF$RUN is provided later in this chapter, following the description of Step 6.

Step 5: Check the Returned Command Status

After you check the returned error code, check the returned severity-code value to determine whether the target program EPF completed successfully. The exact meaning of severity-code is defined by the target program EPF. Typically, if severity-code is 0, the program completed successfully; if severity-code is less than 0, the program encountered problems or unusual conditions but probably completed successfully; if severity-code is greater than 0, the program completed unsuccessfully.

Step 6: Use and Free the Returned Function Value Structure

If you invoked the target program EPF as a function, if code was set to 0 by EPF$RUN, and if rtn-fcn-ptr was not set to the null pointer by the target program EPF, your program should first use the returned function value, for example, by copying it and then return its structure to the pool of available memory. Use FRE$RA to do this, as described later in this chapter.

Error Codes From EPF$RUN

An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT variable. Symbols are provided to allow PL/I-G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings follow. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by EPF$RUN, but are not listed.

Note

When you use EPF$RUN — which itself invokes other EPF$ subroutines — an error code returned by any of those subroutines is returned by EPF$RUN. Therefore, consult the lists of error codes returned by EPFMAP, EPFALLC, EPF$INIT, EPF$INVK, and EPF$DEL, later in this chapter, for information on additional error codes returnable by EPF$RUN.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$EOF	1	End of file. This error indicates a file that has been truncated by FIX_DISK during system maintenance procedures. You must replace the program with a backup copy.
E$UNOP	3	Unit not open. There is no file open on unit. You must open the target program EPF for VMFA-read before calling EPF$RUN to invoke the EPF.
E$BKEY	28	Bad key in call. You are not passing a valid key value to EPF$RUN.
E$BUNT	29	Bad unit number. You are not passing a valid unit value to EPF$RUN.
E$ROOM	55	No room. You cannot invoke the EPF because there is insufficient dynamic storage available to allocate internal EPF information. Use the LIST_EPF and REMOVE_EPF commands to remove inactive EPFs, thereby freeing up dynamic storage.
E$NMTS	106	No more temp segments. You cannot invoke the EPF because you would exceed your limit on dynamic segments. This limit is

		displayed using the LIST_LIMITS command. You should use the LIST_EPF and REMOVE_EPF commands to remove inactive EPFs, thereby freeing up dynamic segments, and attempt to run your program again. If you need more dynamic segments, contact your System Administrator.
E$NMVS	107	No more VMFA segments. You cannot invoke the EPF because there are insufficient segments. The condition may be temporary, in which case an attempt to invoke the target EPF later might succeed. If the condition recurs, consult your System Administrator about increasing the number of VMFA segments on your system (by changing the NVMFS configuration directive in the system startup file).
E$BVER	158	Incorrect version number. Typically, this error means that the function invoked by the call to EPF$RUN returned a structure containing an invalid version number. Alternatively, this error means that the version number of the EPF itself is invalid. In both cases, the fault is in the target EPF, not the calling program.

THE EPF$INVK SUBROUTINE

The EPF$INVK subroutine provides a more controlled, step-by-step interface to the invocation of a program EPF than does the EPF$RUN subroutine. In most ways, however, the use of EPF$INVK is identical to the use of EPF$RUN. This section concentrates on the differences in the use of these two subroutines.

The EPF$INVK subroutine is used in the following manner:

1. The calling program opens the program EPF file to be invoked.

2. The calling program calls EPF$MAP to map the EPF to memory, passing the file unit number of the opened program EPF file and obtaining an EPF identifier for use with the other EPF$ subroutines (except for EPF$RUN, described above).

3. The calling program closes the program EPF.

4. The calling program optionally calls EPF$CPF, passing it the EPF identifier to obtain information on the EPF, such as its selection of command processing features.

5. The calling program calls EPF$ALLC to allocate the linkage areas for the EPF.

6. The calling program calls EPF$INIT to initialize the linkage areas for the EPF.

7. The calling program calls EPF$INVK to invoke the program EPF.

8. After the EPF$INVK subroutine completes, the calling program checks the returned error code to determine whether the program EPF was successfully invoked by EPF$INVK.

9. If the error code from EPF$INVK is 0, the calling program uses the information returned by EPF$INVK to determine whether the program EPF completed successfully or unsuccessfully, and optionally to access the returned text string (if the program EPF was invoked as a function).

10. If the error code from EPF$INVK is 0, and the calling program invoked the program EPF as a function, the calling program uses the FRE$RA subroutine to return the memory used to store the returned text string to the free memory pool.

11. The calling program calls EPF$DEL to remove the program EPF from memory.

For repeated invocations of the same program EPF, repeat Steps 6 through 10. Because avoiding Steps 1 through 5 and Step 11 for subsequent invocations of an EPF saves time, the use of EPF$INVK is sometimes preferred over the use of EPF$RUN.

Note that Step 2 and Steps 4 through 6 correspond to calling the EPF$RUN subroutine with a key value of k$restore_only as described earlier in this chapter. You may choose to use EPF$RUN rather than EPF$MAP, EPF$ALLC, and EPF$INIT, if that is more appropriate for your application. After calling EPF$RUN with the k$restore_only key, close the program EPF file as described in Step 3, then continue with Step 7 of the above procedure to invoke the EPF.

The eleven steps are described below. Steps peculiar to the use of EPF$INVK are described in detail; steps identical to the use of EPF$RUN refer back to the descriptions given in that section.

Step 1: Open the EPF File

This step is identical to Step 1 as described in the section entitled The EPF$RUN Subroutine, earlier in this chapter.

Step 2: Invoke EPF$MAP

The calling program calls EPF$MAP to map the EPF to memory, passing the file unit number of the opened program EPF file and obtaining an EPF identifier for use with the other EPF$ subroutines (except for EPF$RUN, described above). This corresponds to Phase 4 of the life of an EPF, as described in Volume I of this series.

Figure 4-3 illustrates the calling sequence for the EPF$MAP subroutine.

The EPF$MAP subroutine may be used to map either a program EPF or a library EPF. (Although this chapter does not describe the use of EPF$ subroutines on library EPFs, most of them work identically with library EPFs as they do with program EPFs. The exception is EPF$INVK, which supports only the invocation of program EPFs.)

The arguments in the EPF$MAP calling sequence are described next.

The Key: Specify either k$any or k$copy for key. (The value k$dbg is used only by DBG, Prime's source-level debugger. You may use it, but it only increases the amount of virtual memory used by an EPF compiled with the -DEBUG option, without providing any additional functionality.)

The k$any key is most often used, because it specifies that the EPF is to be mapped to any available segments. The procedure (PROC) segments of a mapped EPF cannot be modified by a user, because they may be shared between users by PRIMOS.

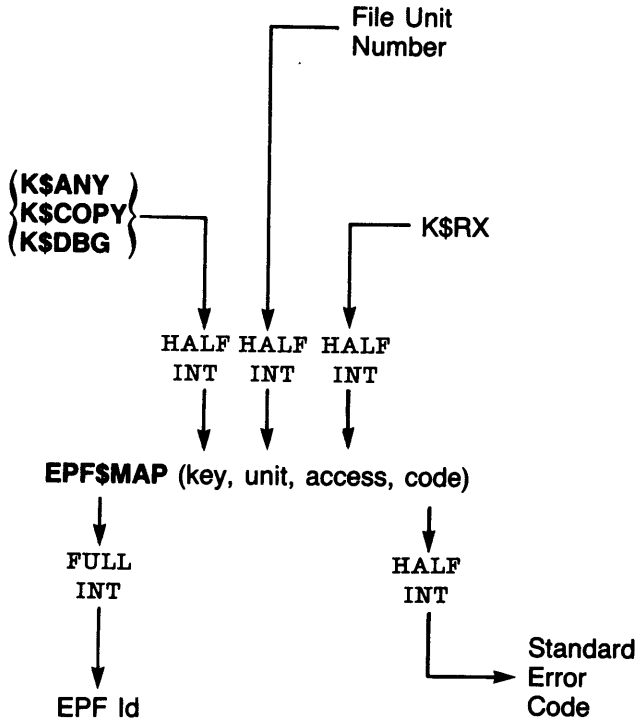
The k$copy key is used when the invoking program intends to modify the procedure (PROC) segments of the EPF. Instead of mapping the procedure segments to memory, k$copy causes EPF$MAP to copy their contents into memory as for static-mode programs. Use the k$copy key if you plan to set breakpoints in an EPF via VPSD, for example.

The File Unit Number: Pass the file unit number of the EPF in unit. This is the unit on which your program opened the EPF runfile for VMFA-read in Step 1. Once you have called EPF$MAP, you can close this unit.

The Segment Access: Pass k$rx in access. This represents the desired segment access. Only one other value is allowed in access, the value k$r. However, both k$rx and k$r result in the same effective segment access — read and execute access. Therefore, always use k$rx access in case k$r is someday redefined to mean something different (such as read-only access).

The Error Code: A standard error code is returned in code. Possible errors codes are summarized later in this chapter.

Map an EPF to Memory



Calling Sequence of EPF$MAP
Figure 4-3

The EPF Identifier: The returned FULL INT value is an identifier of the mapped EPF that your program passes to subsequent EPF$ subroutines to identify the EPF.

Step 3: Close the EPF File

This step is identical with Step 3 described in the section entitled THE EPF$RUN SUBROUTINE, earlier in this chapter.

Step 4: Invoke EPF$CPF (Optional)

The calling program may call EPF$CPF, passing it the EPF identifier, in order to obtain information on the specified EPF, such as its selection of command processing features.

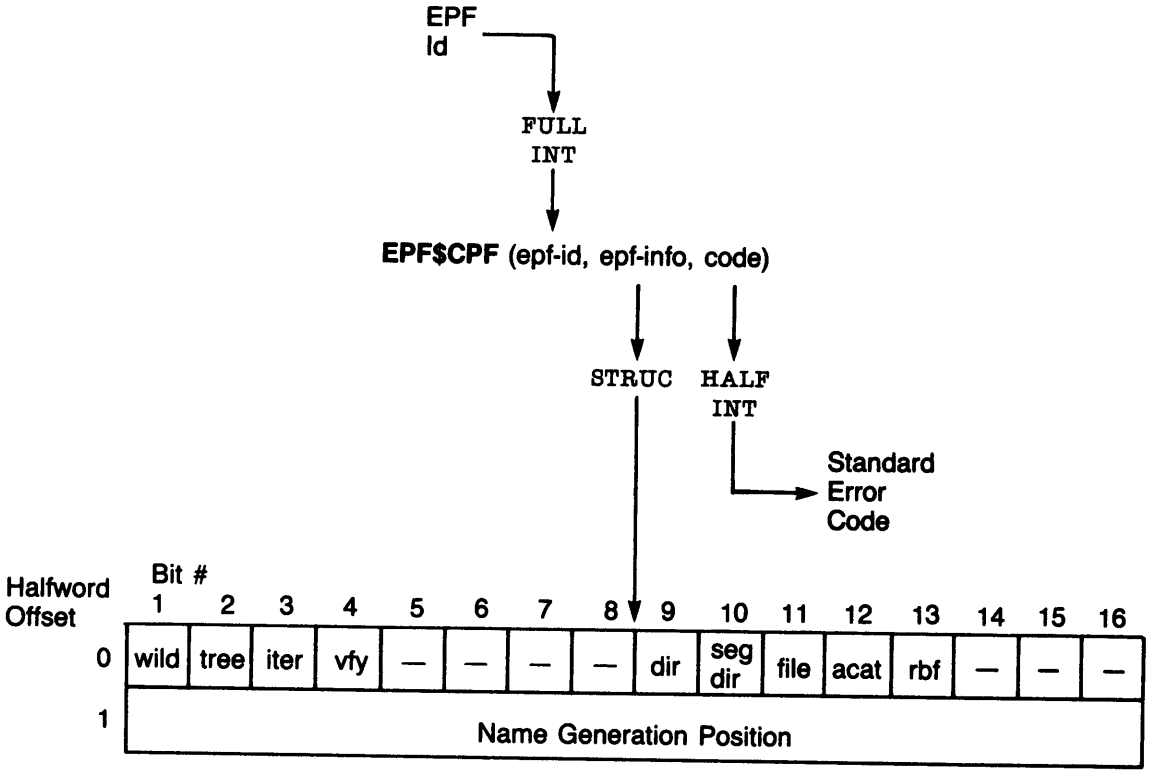
Figure 4-4 illustrates the calling sequence of the EPF$CPF subroutine.

The epf-id and code arguments have the obvious meanings. The epf-info structure, which may be used by your program to select valid command processing features, has the following declaration in PL/I-G:

```
dcl 1 epf_info,  /* EPF info data structure */
      2 command_flags,
        3 wildcards bit(1),  /* Enable wildcards. */
        3 treewalks bit(1),  /* Enable treewalks. */
        3 iteration bit(1),  /* Enable iteration. */
        3 verify bit(1),  /* Verify wildcard selections. */
        3 reserved bit(4) /* Ignore. */
        3 file_types,
          4 directory bit(1),  /* Select directories. */
          4 segdir bit(1),  /* Select segment directories. */
          4 file bit(1),  /* Select files. */
          4 acat bit(1),  /* Select access categories. */
          4 rbf bit(1),  /* Select RBF files. */
          4 reserved bit(3),  /* Ignore. */
      2 name_generation_position fixed bin(15);  /* Token #. */
```

For wildcards, treewalks, and iteration, a bit set to 1 indicates that PRIMOS is to perform the corresponding function. For example, if wildcards is '1'B, PRIMOS intercepts a specification of @@ and expands the command line to several command lines, one for each file system object in the directory (as limited by the object selection in file_types). If wildcards is '0'B instead, PRIMOS passes a specification of @@ to the program EPF without modification, and performs no expansion.

Obtain Information on EPF



Calling Sequence of EPF$CPF
Figure 4-4

The verify bit is the default setting of the -VERIFY or -NO_VERIFY (-VFY or -NVFY) options. When '1'B, the default is -VERIFY; when '0'B, the default is -NO_VERIFY. Actual verification takes place only when wildcards are being processed by the command processor — that is, when wildcards is set to '1'B and the command line contains an actual wildcard specification.

The file_types bits indicate the default settings of the -FILE, -SEGMENT_DIRECTORY (-SEGDIR), -DIRECTORY (-DIR), -ACCESS_CATEGORY (-ACAT), and -RBF options. A bit set to '1'B indicates that the corresponding file type is to be processed.

Note that all file_type bits except the -RBF bit are additive. That is, setting both -FILE and -SEGDIR processes both types. The -RBF bit, however, determines the nature of the other file types. For example, if -SEGDIR and -RBF are both specified, then only RBF segment directories are processed.

The file_types bits are used only during wildcard processing, as with the verify bit. For example, if the command RESUME MYPROG XYZ is given, MYPROG is invoked for the file system object named XYZ even if XYZ is a directory and the directory bit is reset to '0'B. However, if the command RESUME MYPROG XYZ@@ is given (and the wildcards bit is '1'B), the XYZ directory is not selected if directory is '0'B, because wildcard processing is taking place.

The name_generation_position variable is an integer that specifies which of the tokens that follow the program or command name is to be used as the name generation source pattern. Normally, this variable is set to 1, meaning that the first token after the RESUME MYPROG tokens is to be used as the source pattern. For example, the command line

```
RESUME MYPROG FOO BAR ==
```

produces an effective command line of:

```
RESUME MYPROG FOO BAR FOO
```

However, if name_generation_position is 2, the second token is used instead. For example, given the same command line above, the effective command line produced when name_generation_position is 2 is:

```
RESUME MYPROG FOO BAR BAR
```

The token count for a program EPF installed in CMDNC0 also begins immediately after the program name. Therefore, the following two command lines always produce the same result with regard to name generation pattern processing.

```
MYPROG A B ==

RESUME CMDNC0>MYPROG A B ==
```

Step 5: Invoke EPF$ALLC

The calling program now calls EPF$ALLC to allocate the linkage areas for the EPF, passing the EPF identifier. This step corresponds to Phase 5 of the life of an EPF.

Figure 4-5 illustrates the calling sequence of the EPF$ALLC subroutine.

The epf-id and code arguments have the usual meanings.

Step 6: Invoke EPF$INIT

The calling program calls EPF$INIT to initialize the linkage areas for the EPF, passing the EPF identifier. This step corresponds to Phase 6 of the life of an EPF.

Figure 4-6 illustrates the calling sequence of the EPF$INIT subroutine.

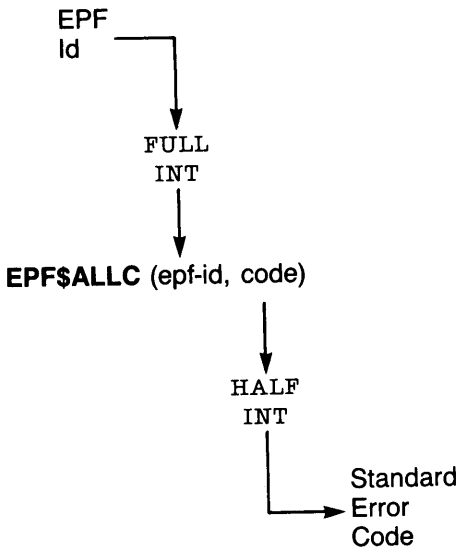
The epf-id and code arguments have the usual meanings.

The key argument specifies whether a complete initialization is to be performed. The first time EPF$INIT is called for an EPF that has just had its linkage allocated via EPF$ALLC, key must be set to k$initall, which specifies complete initialization. After calling EPF$INVK, in the next step, a subsequent invocation of the program requires only a call to EPF$INIT with a key of k$reinit to reinitialize only certain portions of the linkage areas for the EPF before calling EPF$INVK again.

Specifically, while a key of k$initall specifies complete initialization of the linkage areas, a key of k$reinit specifies that only faulted IPs (dynamic links) and static data are to be reinitialized. ECBs, static IPs, and other nonfaulted IPs are not reinitialized. Once initialized, they do not need to be initialized again unless the program modifies them during execution (which is considered poor programming practice).

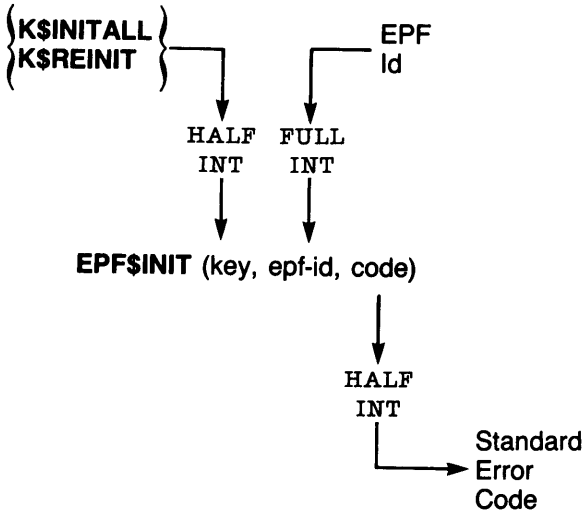
If a program being invoked by your program seems to fail in strange ways after the first invocation, have your program use the k$initall key exclusively to see if the problem is caused by the invoked program — it might be modifying linkage data that should not be modified once it has been initialized by EPF$INIT.

Allocate Linkage Areas for EPF



Calling Sequence of EPF$ALLC
Figure 4-5

Initialize Linkage Areas for EPF



Calling Sequence of EPF$INIT
Figure 4-6

Step 7: Invoke EPF$INVK

The calling program calls EPF$INVK to invoke the program EPF, passing the EPF identifier. This step corresponds to Phase 7 of the life of an EPF.

Figure 4-7 illustrates the calling sequence for the EPF$INVK subroutine.

The epf-id and code arguments have the usual meanings. The remaining arguments correspond precisely to the same arguments to the EPF$RUN subroutine, described earlier in this chapter. In fact, as with EPF$RUN, the latter five arguments may be omitted if the main entrypoint of the target program EPF does not accept any arguments.

Step 8: Check the Returned Error Code

This step is identical to Step 4 in the section entitled THE EPF$RUN SUBROUTINE, earlier in this chapter.

Step 9: Check the Returned Command Status

This step is identical to Step 5 in the section entitled THE EPF$RUN SUBROUTINE, earlier in this chapter.

Step 10: Use and Free the Returned Function Value Structure

This step is identical to Step 4 in the section entitled THE EPF$RUN SUBROUTINE, earlier in this chapter.

Step 11: Invoke EPF$DEL

The calling program calls EPF$DEL to remove the program EPF from memory, passing the EPF identifier. This step corresponds to Phase 10 of the life of an EPF.

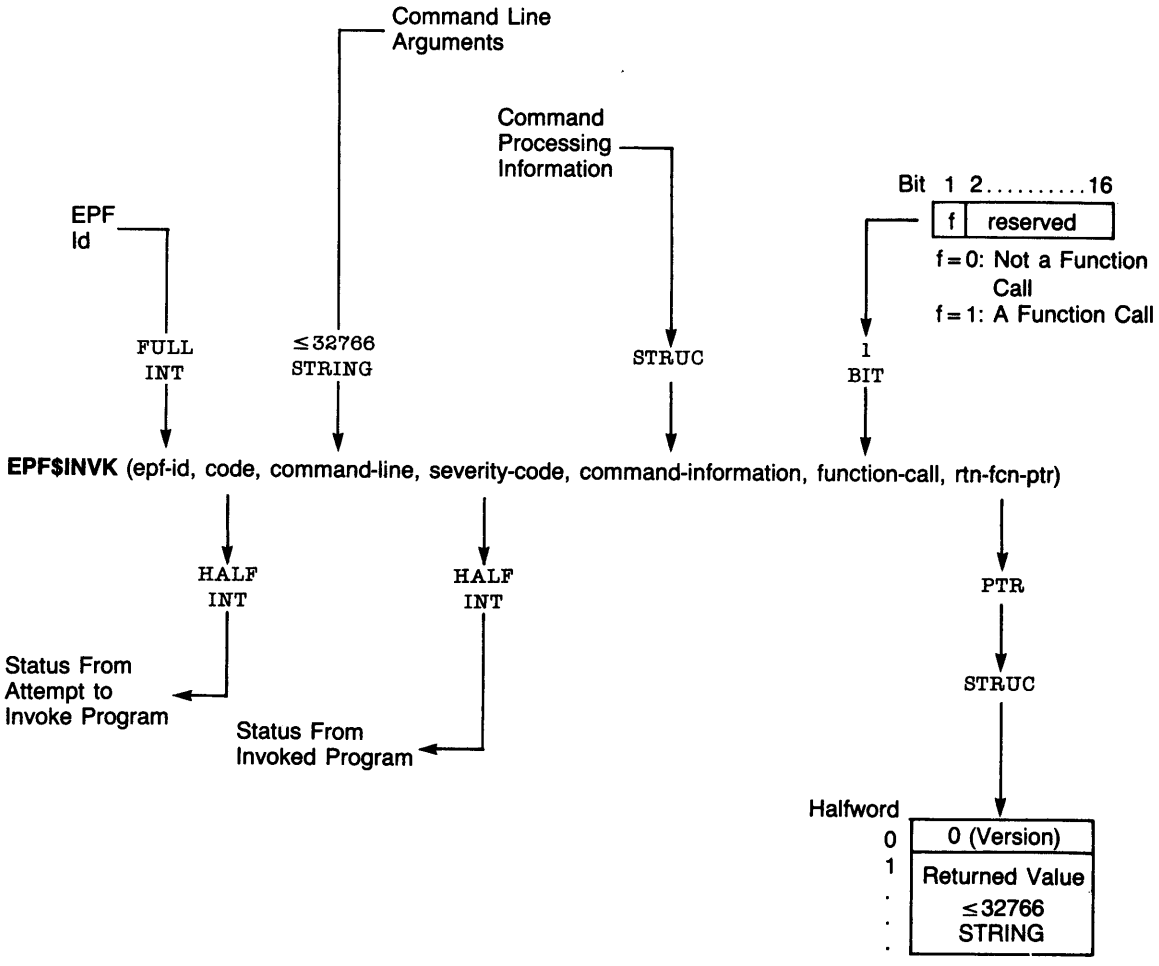
Figure 4-8 illustrates the calling sequence of the EPF$DEL subroutine.

The epf-id and code arguments have the usual meanings.

If the EPF is still in use by this process, such as when a user types Control-P while the program is executing, then the EPF is not removed, and an error code (e$swpr) is returned in code.

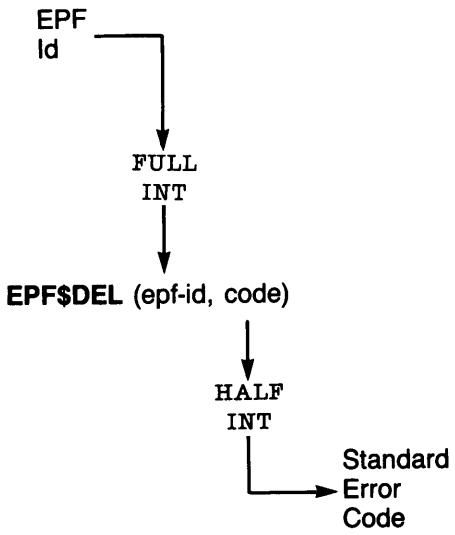
The EPF is not actually removed from the system's virtual memory if other users have the EPF mapped to their memory. However, it is

Invoke a Program EPF



Calling Sequence of EPF$INVK
Figure 4-7

Remove an EPF From Memory



Calling Sequence of EPF$DEL
Figure 4-8

unmapped from the calling user's memory, and is removed from the system's virtual memory when the last user unmaps it from his or her memory.

Error Codes From EPF$ Subroutines

All of the EPF$ subroutines may encounter errors. In addition, opening a file for VMFA-read may result in an error that pertains specifically to the VMFA mechanism, rather than the file access mechanism. An output argument, code, informs the calling program of the success or failure of the operation. This argument is a HALF INT value. Symbols are provided to allow PL/I-G, FORTRAN, Pascal, and PMA programs to substitute mnemonic keywords for numeric values.

If code is 0, the operation was entirely successful. Otherwise, code has one of many values. Typical values and their meanings are listed for each EPF$ subroutine. Not all possible error codes are listed; for example, PRIMENET-related error codes such as E$RLDN (The remote line is down) may be returned by one or more of these subroutines, but are not listed.

Error Codes Involving the K$VMR Key: Error codes specific to opening a file for VMFA-read (using the k$vmr key) are listed below. Other error codes applying to opening files in general may also be returned.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$NRIT	10	The user has insufficient access to open the target file for VMFA-read. Currently, Read access to the file is required.
E$NDAM	109	The target object is not a DAM file; this error code is also returned if an attempt is made to open the cache directory by specifying the k$curr value for the filename or by specifying a null pathname.

Error Codes From EPF$MAP: Error codes that may be returned by EPF$MAP are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$UNOP	3	The unit specified in unit is not open.

E$BPAR	6	An invalid segment access has been specified in access. It must be either k$rx or k$r.
E$BKEY	28	The value of key is invalid.
E$BUNT	29	The value specified in unit is an invalid file unit number.
E$NMVS	107	There are not enough VMFA segments in the system to accommodate the EPF. If this errors persists, contact your System Administrator, who may wish to increase the number of VMFA segments on your system (via the NVMFS configuration directive in the system configuration file).
E$NMTS	108	There are no more temporary segments available into which the EPF procedure segments can be copied.
E$NDAM	109	The file open on unit is not a DAM file.
E$NOVA	110	The file open on unit is not open for VMFA-read. It must be opened using the k$vmr key.
E$BVER	158	Invalid EPF version. The file open for VMFA-read is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS.
E$EPFT	217	The file open for VMFA-read on the file unit is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS later than Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS.
E$EPFL	222	The EPF file is too large for the current EPF implementation. More segments are required by the EPF than are supported by the current revision of PRIMOS. If you are using the -DEBUG option, recompile the program without the option to reduce its size. Alternatively, consider splitting the program up into smaller pieces, such as one program EPF and one or more library EPFs.

Error Codes From EPF$CPF: Error codes that may be returned by EPF$CPF are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.

Error Codes From EPF$ALLC: Error codes that may be returned by EPF$ALLC are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPF$MAP, this error is not likely to occur when calling EPF$ALLC unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS later than Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPF$MAP, this error is not likely to occur when calling EPF$ALLC unless it is called out of sequence.
E$ILTD	219	The EPF contains an invalid linkage descriptor. The problem is not with the calling program; this error usually indicates a corrupted EPF file.

Error Codes From EPF$INIT: Error codes that may be returned by EPF$INIT are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BKEY	28	Either the key argument is invalid (not k$initall or k$reinit), or the k$reinit key is specified but the linkage areas for the EPF have not yet been fully initialized (by specifying the k$initall key in a call to EPF$INIT).
E$BARG	71	The EPF$ALLC has not yet been successfully called to allocate linkage areas for this EPF.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPF$MAP and EPF$ALLC, this error is not likely to occur when calling EPF$INIT unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS later than Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPF$MAP and EPF$ALLC, this error is not likely to occur when calling EPF$INIT unless it is called out of sequence.
E$ILTD	219	The EPF contains an invalid linkage descriptor. The problem is not with the calling program; this error usually indicates a corrupted EPF file.
E$ILTE	220	The EPF contains an invalid linkage descriptor. The problem is not with the calling program; this error usually indicates a corrupted EPF file.

Error Codes From EPF$INVK: Error codes that may be returned by EPF$INVK including any codes that may be returned by EPF$DEL in addition to those listed below.

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, and EPF$INIT, this error is not likely to occur when calling EPF$INVK unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, and EPF$INIT, this error is not likely to occur when calling EPF$INVK unless it is called out of sequence.
E$ECEB	221	The command environment breadth limit has been reached; the currently running program can call no more programs. Use the LIST_LIMITS command to display command environment limits, or use the CE$BRD subroutine to determine the command environment breadth limit from within a program.

Error Codes From EPF$DEL: Error codes that may be returned by EPF$DEL are:

Keyword	Value	Meaning
<ok>	0	The operation was successful.
E$BPAR	6	The epf-id passed represents an EPF that is no longer mapped to memory.
E$BVER	158	Invalid EPF version. The EPF is either a corrupted EPF, not an EPF, or an EPF generated by a future revision of PRIMOS that is not supported by the current revision of PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK, this error is not likely to occur when calling EPF$DEL unless it is called out of sequence.
E$EPFT	217	The EPF is not a valid EPF. Either the file contains a corrupted EPF or is not an EPF at all, or the file contains an EPF generated by a revision of PRIMOS beyond Rev. 19.4 that is not recognized by Rev. 19.4 PRIMOS. Because this condition is checked by EPFMAP, EPFALLC, EPF$INIT, and EPF$INVK, this error is not likely to occur when calling EPF$DEL unless it is called out of sequence.
E$SWPR	225	The EPF is suspended by this user process, and hence cannot be unmapped from memory. This error code is returned if a program attempts to call EPF$DEL to unmap itself.

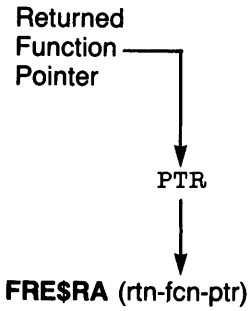
THE FRE$RA SUBROUTINE

After calling CP$, EPF$RUN, or EPF$INVK to invoke a function, and after making use of the returned function value, your program must call FRE$RA to free the memory used to hold the returned function value. (Call FRE$RA only if the function your program invoked actually returned a function value.)

Figure 4-9 illustrates the calling sequence of FRE$RA. Simply pass the returned function pointer (rtn-fcn-ptr).

For information on how returned function values are set, see Chapter 3, including the descriptions of the ALS$RA and ALC$RA subroutines.

Free a Returned Function Value Structure



Calling Sequence of FRE$RA
Figure 4-9

SAMPLE PROGRAMS

The first sample program is called SLOW_INVOKE. It takes an EPF name and command arguments for the EPF as arguments to the program, and it then performs each step associated with executing the target EPF. After each step, it pauses so that the user may use the LIST_EPF -DETAIL command to see how far it has gotten. Although not necessarily a useful example by itself, this program does illustrate how each step is performed, and also shows the PL/I-G declarations for the appropriate subroutines and structures.

```
slow_invoke: proc(x_command_line,code,command_state,command_flags,
      return_function_ptr);

dcl x_command_line char(1024) var,
    code fixed bin(15),
    1 command_state,
      2 com_name char(32) var,
      2 version fixed bin(15),
      2 vcb_ptr ptr,
      2 cp_iter_info,
         3 mod_after_date fixed bin(31),
         3 mod_before_date fixed bin(31),
         3 bk_after_date fixed bin(31),
         3 bk_before_date fixed bin(31),
         3 type_dir bit(1),
         3 type_segdir bit(1),
         3 type_file bit(1),
         3 type_acat bit(1),
         3 type_rbf bit(1),
         3 mbz1 bit(11),
         3 verify_sw bit(1),
         3 botup_sw bit(1),
         3 mbz2 bit(14),
         3 walk_from fixed bin(15),
         3 walk_to fixed bin(15),
         3 in_iteration bit(1),
         3 in_wildcard bit(1),
         3 in_treewalk bit(1),
         3 mbz3 bit(13),
    1 command_flags,
      2 command_function_call bit(1),
      2 mbz bit(15),
    return_function_ptr ptr;

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl epf_unit fixed bin(15),
    epf_id fixed bin(31),
    epf_filename char(128) var,
    i fixed bin(15),
    command_line char(1024) var,
```

```
        epf_command_line char(1024) var,
        basename char(32) var,
        suffix_used fixed bin(15),
        type fixed bin(15),
        command_status fixed bin(15);

    dcl errpr$ entry(fixed bin(15),fixed bin(15),char(80),
          fixed bin(15),char(80),fixed bin(15)),
        srsfx$ entry(fixed bin(15),char(128) var,fixed bin(15),
          fixed bin(15),fixed bin(15),char(32) var,char(32) var,
          fixed bin(15),fixed bin(15)),
        clo$fu entry(fixed bin(15),fixed bin(15)),
        tnou entry(char(80),fixed bin(15)),
        epf$map entry(fixed bin(15),fixed bin(15),fixed bin(15),
          fixed bin(15)) returns(fixed bin(31)),
        epf$allc entry(fixed bin(31),fixed bin(15)),
        epf$init entry(fixed bin(15),fixed bin(31),fixed bin(15)),
        epf$invk entry(fixed bin(31),fixed bin(15),char(1024) var,
          fixed bin(15),
          1, 2 char(32) var,
             2 fixed bin(15),
             2 ptr,
             2, 3 fixed bin(31),
                3 fixed bin(31),
                3 fixed bin(31),
                3 fixed bin(31),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(11),
                3 bit(1),
                3 bit(1),
                3 bit(14),
                3 fixed bin(15),
                3 fixed bin(15),
                3 bit(1),
                3 bit(1),
                3 bit(1),
                3 bit(13),
           1, 2 bit(1),
              2 bit(15),
           ptr),
        epf$del entry(fixed bin(31),fixed bin(15));

    command_line=trim(x_command_line,'11'b);
    i=index(command_line,' ');

    if i=0 & command_line=''
       then do;
            code=e$ivcm;
            call errpr$(k$irtn,code,'Specify EPF filename',20,
                 'SLOW_INVOKE',11);
```

```
        return;
        end;

if i=0
   then do;
        epf_filename=command_line;
        epf_command_line='';
        end;
   else do;
        epf_filename=substr(command_line,1,i-1);
        epf_command_line=trim(substr(command_line,i+1),'11'b);
        end;

call srsfx$(k$getu+k$vmr,epf_filename,epf_unit,type,1,'.RUN',
     basename,suffix_used,code);
if code^=0
   then do;
        call errpr$(k$irtn,code,(epf_filename),
             length(epf_filename),'SLOW_INVOKE',11);
        return;
        end;

call tnou('SRSFX$ complete',15);
call pause_me;

epf_id=epf$map(k$any,epf_unit,k$rx,code);
call clo$fu(epf_unit,i);
if code^=0
   then do;
        call errpr$(k$irtn,code,'Mapping '||epf_filename,
             length(epf_filename)+8,'SLOW_INVOKE',11);
        return;
        end;

call tnou('EPF$MAP complete',16);
call pause_me;

call epf$allc(epf_id,code);
if code^=0
   then do;
        call clo$fu(epf_unit,i);
        call errpr$(k$irtn,code,'Allocating '||epf_filename,
             length(epf_filename)+11,'SLOW_INVOKE',11);
        return;
        end;

call tnou('EPF$ALLC complete',17);
call pause_me;

call epf$init(k$initall,epf_id,code);
if code^=0
   then do;
        call clo$fu(epf_unit,i);
        call errpr$(k$irtn,code,'Initializing '||epf_filename,
```

```
                length(epf_filename)+13,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$INIT complete',17);
    call pause_me;

    command_status=0;
    command_state.com_name=basename;
    call epf$invk(epf_id,code,epf_command_line,command_status,
         command_state,command_flags,return_function_ptr);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Invoking '||epf_filename,
                 length(epf_filename)+10,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$INVK complete',17);
    call pause_me;

    call epf$del(epf_id,code);
    if code^=0
       then do;
            call clo$fu(epf_unit,i);
            call errpr$(k$irtn,code,'Removing '||epf_filename,
                 length(epf_filename)+9,'SLOW_INVOKE',11);
            return;
            end;

    call tnou('EPF$DEL complete',16);
    call pause_me;

    code=command_status;
    return;

    pause_me: proc;

    dcl pause_ char(32) var static init('PAUSE$');

    dcl signl$ entry(char(32) var,ptr options(short),fixed bin(15),
         ptr options(short),fixed bin(15),bit(1) aligned);

    call signl$(pause_,null(),0,null(),0,'1'b);

    end;  /* pause_me: proc */

    end;
```

The next sample program, called DISPLAY_EPF_INFO, displays command processing information for an EPF by mapping it to memory, calling EPF$CPF, and then removing the EPF from memory. It illustrates how to process the information returned by EPF$CPF.

```
display_epf_info: proc(command_line,code,command_state,
     command_flags,return_function_ptr);

dcl command_line char(1024) var,
    code fixed bin(15),
    1 command_state,
      2 com_name char(32) var,
      2 version fixed bin(15),
    1 command_flags,
      2 command_function_call bit(1),
      2 mbz bit(15),
    return_function_ptr ptr;

%include 'SYSCOM>ERRD.INS.PL1';
%include 'SYSCOM>KEYS.INS.PL1';

dcl epf_unit fixed bin(15),
    epf_id fixed bin(31),
    epf_filename char(128) var,
    i fixed bin(15),
    basename char(32) var,
    suffix_used fixed bin(15),
    type fixed bin(15);

dcl errpr$ entry(fixed bin(15),fixed bin(15),char(80),
      fixed bin(15),char(80),fixed bin(15)),
    srsfx$ entry(fixed bin(15),char(128) var,fixed bin(15),
      fixed bin(15),fixed bin(15),char(32) var,char(32) var,
      fixed bin(15),fixed bin(15)),
    clo$fu entry(fixed bin(15),fixed bin(15)),
    tnou entry(char(80),fixed bin(15)),
    epf$map entry(fixed bin(15),fixed bin(15),fixed bin(15),
      fixed bin(15)) returns(fixed bin(31)),
    epf$del entry(fixed bin(31),fixed bin(15));

if command_line=''
   then do;
        code=e$ivcm;
        call errpr$(k$irtn,code,'Specify EPF filename',20,
             (com_name),length(com_name));
        return;
        end;

epf_filename=command_line;

call srsfx$(k$getu+k$vmr,epf_filename,epf_unit,type,1,'.RUN',
     basename,suffix_used,code);
if code^=0
```

```
      then do;
           call errpr$(k$irtn,code,(epf_filename),
                length(epf_filename),(com_name),length(com_name));
           return;
           end;

   epf_id=epf$map(k$any,epf_unit,k$rx,code);
   call clo$fu(epf_unit,i);  /* Close the unit. */
   if code^=0
      then do;
           call errpr$(k$irtn,code,'Mapping '||epf_filename,
                length(epf_filename)+8,(com_name),length(com_name));
           return;
           end;

   call say_nl(trim(char(epf_id),'11'b));

   call show_epf_info(epf_id);  /* Display the information. */

   call epf$del(epf_id,code);
   if code^=0 & code^=e$swpr
      then do;
           call clo$fu(epf_unit,i);
           call errpr$(k$irtn,code,'Removing '||epf_filename,
                length(epf_filename)+9,(com_name),length(com_name));
           return;
           end;
      else if code=e$swpr
              then call say_nl('(Still suspended by this process.)');

   code=0;
   return;

   show_epf_info: proc(epf_id);

   dcl epf_id fixed bin(31);

   dcl code fixed bin(15),
       1 epf_info,  /* EPF info data structure */
         2 command_flags,
           3 wildcards bit(1),
           3 treewalks bit(1),
           3 iteration bit(1),
           3 verify bit(1),
           3 reserved bit(4),
           3 file_types,
             4 directory bit(1),
             4 segdir bit(1),
             4 file bit(1),
             4 acat bit(1),
             4 rbf bit(1),
             4 reserved bit(3),
         2 name_generation_position fixed bin(15);
```

```
dcl epf$cpf entry(fixed bin(31),
    1, 2, 3 bit(1),
          3 bit(1),
          3 bit(1),
          3 bit(1),
          3 bit(4),
          3,
             4 bit(1),
             4 bit(1),
             4 bit(1),
             4 bit(1),
             4 bit(1),
             4 bit(3),
       2 fixed bin(15),
       fixed bin(15));

/* Call EPF$CPF to get the information. */

call epf$cpf(epf_id,epf_info,code);
if code^=0 then call errpr$(k$irtn,code,'Calling EPF$CPF',15,
                      (com_name),length(com_name));
   else do;

/* Command processing info. */

        call say_nl('');
        call say_nl('Info on '||epf_filename||':');
        call say_nl('');

        call say('Command processing:');

        if epf_info.wildcards then call say(' wild');
        if epf_info.treewalks then call say(' tree');
        if epf_info.iteration then call say(' iter');
        if epf_info.verify then call say(' vfy');

        call say_nl('');
        call say('Object selection:');

        if epf_info.directory then call say(' dir');
        if epf_info.segdir then call say(' segdir');
        if epf_info.file then call say(' file');
        if epf_info.acat then call say(' acat');
        if epf_info.rbf then call say(' rbf');

        call say_nl('');
        call say_nl('Name generation position: '
             ||trim(char(name_generation_position),'11'b));

        call say_nl('');

        end;

 end;  /* show_epf_info: proc */
```

```
say: proc(text);

dcl text char(*) var;

dcl tnoua entry(char(*),fixed bin(15));

call tnoua((text),length(text));

end; /* say: proc */

say_nl: proc(text);

dcl text char(*) var;

dcl tnou entry(char(*),fixed bin(15));

call tnou((text),length(text));

end; /* say: proc */

end;
```

IF A PROGRAM INVOKES ITSELF

A program may invoke itself recursively, either directly by calling itself using CP$, EPF$RUN, or EPF$INVK, or indirectly by calling another program or collection of programs that ultimately call the original program.

A program invoking itself recursively via CP$, EPF$RUN, or EPF$INVK, whether directly or indirectly, does not necessarily produce the same results as it does when it calls itself by invoking its own main entrypoint. In both cases, dynamic storage is allocated and initialized for each invocation. However, static storage is allocated only during program invocation; it is allocated for all procedures in that program each time the program is invoked. Once the program is running, no additional static storage is allocated by PRIMOS.

PRIMOS allocates and initializes one copy of static storage per program invocation. Static storage includes COMMON and STATIC EXTERNAL areas except for those explicitly named using the SYMBOL subcommand of BIND. In addition, static storage contains subroutine linkage pointers, static data (SAVE or DATA in FORTRAN, STATIC in PL/I), and program constants.

Because PRIMOS separates program invocations so that they cannot destroy one another's data, one program can be invoked, suspended, and reinvoked; and then the original invocation can be continued by issuing the START command. The second invocation of the program does not affect the first invocation of the program; therefore, the results of the first invocation are essentially unchanged.

Of course, if a program makes use of data that is not in static or dynamic storage, such as COMMON or STATIC EXTERNAL storage specified using the SYMBOL command, then separate invocations of the program are not necessarily independent of each other. Other data not in static or dynamic storage includes system objects such as attach points, files, file units, and so on. PRIMOS does not provide a fully recursive command environment, it provides only a separation of per-program data between program invocations. See Chapter 6 for more information on this subject.

Terminal Input and Output

Keep in mind that invoking a command from within a program does not redirect terminal input or output. For example, if you invoke the LD command from within a program, the output from LD is sent to the user terminal, and responses to the —More— prompt are solicited from the user terminal.

Therefore, you may wish to use the COMO$$ subroutine or the internal PRIMOS command COMOUTPUT to redirect terminal output to a command output file. To redirect terminal input to a command input file written by your program, you may use COMI$$ or the internal PRIMOS command COMINPUT; alternatively, when supported by the command, you may specify an option indicating how to substitute for terminal input. (For example, LD accepts a -NO_WAIT option, which specifies that —More— prompts are not to be issued.)

Few functions perform any terminal I/O. Some allow the invoking program to specify command line options that disable or redirect terminal I/O.

5

The Command Processor Stack

This chapter describes the command processor stack and how to examine and manipulate it.

WHAT THE COMMAND PROCESSOR STACK IS USED FOR

The command processor stack is used by the command processor to hold information that pertains to a particular command level or to a particular program invocation. In addition, the command processor stack is used by program EPFs to hold stack frames for procedures.

In fact, the command processor itself consists of a collection of procedures that use the command processor stack for their stack frames.

A procedure that uses a particular stack is said to be executing on that stack. Therefore, a program EPF that is invoked by the command processor executes on the command processor stack.

Subroutines in a library EPF usually execute on the same stack as their calling program. Because a library EPF is a collection of entrypoints, different subroutines in a library EPF may execute on different stacks; the library EPF itself does not execute on a particular stack.

Therefore, many of the programs that you develop, implement, debug, and test will use the command processor stack. You will find it useful to know how to examine and manipulate the stack.

COMMAND LEVELS

As explained in Chapter 1, a command level consists of the invocation of the listener, which displays the OK, or ER! prompt and awaits the input of a user command.

The Listener

A command level is distinguished by the invocation of the listener, whose stack frame on the command processor stack indicates a new command level. Typically, this stack frame is easily distinguished by the conditions for which it has set up on-units. You can see this by issuing the DUMP_STACK command while at command level 1:

```
OK, DUMP_STACK -ON_UNITS
No condition frame exists on the stack. (dmstk)
Backward trace of stack from frame 1 at 6002(3)/6576.

STACK SEGMENT IS 6002.

(1) 006576: Owner=  (LB= 13(0)/20250).
    Called from 13(3)/135505; returns to 13(3)/135515.

(2) 005520: Owner=  (LB= 13(0)/137334).
    Called from 13(3)/134650; returns to 13(3)/134654.

(3) 002212: Owner=  (LB= 13(0)/137334).
    Called from 13(3)/13521; returns to 13(3)/13541.

(4) 001406: Owner=  (LB= 13(0)/20250).
    Called from 13(3)/7233; returns to 13(3)/7245.
    Onunit for "CLEANUP$" is 13(3)/21331.
    Onunit for "STOP$" is 13(3)/21051.
    Onunit for "SUBSYS_ERR$" is 13(3)/21071.

(5) 000640: Owner=  (LB= 13(0)/11220).
    Called from 13(3)/163406; returns to 13(3)/163412.
    Onunit for "CLEANUP$" is 13(3)/12016.
    Onunit for "ANY$" is 13(3)/124163.
    Onunit for "LISTENER_ORDER$" is 13(3)/12056.
    Onunit for "SETRC$" is 13(3)/12036.
    Onunit for "REENTER$" is 13(3)/12076.

(6) 000632: Owner=  (LB= 13(0)/163024).
    Called from 1(0)/163410; returns to 1(0)/0.
OK,
```

In the above example, the stack root is segment '6002. This is, in fact, the command processor stack segment at Rev. 19.4, although it may change in future revisions.

The stack frame labeled frame 6 in the above display is always at the bottom of the command processor stack. Its frame number varies depending on the number of stack frames on the stack at the time the DUMP_STACK command is issued.

The stack frame labeled frame 5 is the listener's stack frame. Notice the long list of conditions that it catches; in particular, it catches the ANY$ condition, which is the default on-unit (called DF_UNIT_).

The listener inputs a command and then invokes the command processor, which is identified in the above sample display as stack frame 4, and which is named STD$CP. (STD$CP stands for "Standard Command Processor.")

Stack frames 3 and 2 belong to the iteration processor, which handles all of the iteration list, treewalk, and wildcard processing along with the corresponding options (such as -MODIFIED_BEFORE and -WALK_FROM). Stack frame 3 belongs to the iteration processor itself, while stack frame 2 belongs to a procedure internal to the processor. Because the DUMP_STACK -ON_UNITS command specifies no iteration, none is performed by the iteration processor.

Stack frame 1 belongs to a procedure internal to STD$CP that executes internal PRIMOS commands; DUMP_STACK is such a command. DUMP_STACK does not display the stack frames it creates, but they are also present during the execution of the command. The internal-command executor that owns stack frame 1 invokes the DUMP_STACK procedure. When DUMP_STACK finishes executing, it returns to the internal-command executor, which returns to the iteration processor. The iteration processor sees no more commands to process (because there is no form of iteration in use) so it returns to the command processor. Because the command processor sees no more commands to process (a semicolon, the command separator, is not present), it returns to the listener, which displays the OK, prompt and awaits another command.

Multiple Invocations of the Listener

More than one invocation of the listener causes multiple command levels to be created. This is what happens when a running program encounters an error such as an illegal instruction or when the user types Control-P. Rather than stop the running program and return to the listener, which would wipe out the stack history of the running program, rendering it unrestartable and difficult to debug, the default on-unit (DF_UNIT_) catches the condition (such as ILLEGAL_INST$ or QUIT$) and invokes the listener again, creating a new command level.

From this new command level, you may issue the START command to attempt to continue the program. This causes the listener to return to its caller, the default on-unit, which itself returns to its caller, the condition-signaling mechanism. The condition-signaling mechanism then returns to its caller, the running program. The instruction being

processed when the interruption occurred is retried, and the program continues running.

Not all interruptions are restartable — some program faults require some form of repair before the program can be restarted. The QUIT$ condition, however, is restartable as described above. What this means to you as a programmer is specified in more detail at the end of this chapter.

THE RDY COMMAND

The RDY command specifies the kind of prompt you wish displayed. When working with the command environment, you will find the RDY command useful because it can tell you what command level you are currently at. In addition, the RDY command can display useful CPU and I/O usage information for your process.

The Long Prompt

You can use the RDY command to display a long prompt that contains a user-selectable prompt followed by the time of day, incremental CPU and I/O times, the command level number, and an indicator of whether a static-mode program is in use at the current level.

For example:

```
OK 15:25:14  245.275  19.103
```

Here, the level number is not displayed because the user is at command level 1. If the user is at command level 2 or above, the display appears as in the following example:

```
OK 15:26:36  5.869  0.118  level 2
```

If a static-mode program is in use at the current level, a plus sign (+) is displayed immediately following the level number:

```
OK 15:26:36  5.869  0.118  level 2+
```

The + sign indicates that a START command will continue execution of the static-mode program rather than restarting the suspended program at command level 1. It also indicates that a RELEASE_LEVEL command will release the static-mode program (meaning that a subsequent START would then start the suspended program at level 1).

Turning On Long Prompts

To turn on long prompts permanently, place the RDY -LONG command in your LOGIN.CPL file.

Tailoring Your Prompts

To specify the prompt you wish displayed, use the various options of RDY as described in the PRIMOS Commands Reference Guide. For example:

```
OK 15:32:22  23.721  0.000  level 2
RDY -READY_LONG 'System SYSA'
System SYSA 15:32:33  0.160  0.000  level 2
```

THE RELEASE_LEVEL COMMAND

The RELEASE_LEVEL command releases command levels. Because a command level is really an invocation of the listener, RELEASE_LEVEL actually performs a nonlocal GOTO to a previous invocation of the listener (unless it is being run by the listener invocation for command level 1, in which case it does nothing).

Generally, there are two reasons to use the RELEASE_LEVEL command:

- To release resources used by one or more levels and by the programs invoked on those levels
- To return to a particular level so that a subsequent START command will continue the appropriate program

Typically, you use RELEASE_LEVEL to achieve one or both of the above effects.

Please note that you cannot follow the RELEASE_LEVEL command with the command separator followed by more commands. For example, the command

```
RELEASE_LEVEL -ALL;CLOSE -ALL
```

does not close any file units.

Instead, you must use:

```
CLOSE -ALL;RELEASE_LEVEL -ALL
```

This restriction is necessary because, as described earlier in this chapter, the command processor itself executes on the command processor stack, and subroutines that perform various forms of iteration are part of the command processor. Therefore, in the first example above, the RELEASE_LEVEL command releases the invocation of the procedure that is prepared to execute the CLOSE -ALL command after RELEASE_LEVEL returns. Note, too, that RELEASE_LEVEL never returns. Instead, it performs a nonlocal goto by signaling a special condition. (The signaling of the condition invokes an on-unit for the listener invocation that is the target of the RELEASE_LEVEL command. It is the on-unit that actually performs the non-local goto.)

Releasing Resources

When you release a particular command level, all resources used by that level are released. The resources are those used by the command processor itself and those used by all EPFs or internal commands that were invoked at that command level. (Certain static-mode program resources cannot be released, as noted below.) Such resources include:

- Stack frames
- Dynamically allocated storage (except for process-class and subsystem-class storage)
- Linkage areas and common storage (depending upon EPF type, but not for static-mode programs)
- Procedure code storage (depending upon EPF type, but not for static-mode programs)
- Any resources (such as file units) freed by on-units for the CLEANUP$ condition

When you release a particular level, all of these resources are freed. In addition, the resources listed above are freed for the level that is the target of the release, except for the stack frames used by the invocation of the listener for that level. Therefore, you can issue the RELEASE_LEVEL command to release to level 1, and resources used by programs invoked on level 1 are freed; however, level 1 itself can be freed only by logging out.

Releasing to Restart a Suspended Program

You may sometimes want to release levels down to a particular level in order to use the START command to continue the execution of a program invoked one level lower. Thus, the following example shows a user giving the command "RELEASE_LEVEL -TO 2," in order to restart a program on level 1.

```
OK, RDY -LONG
OK 17:50:36  131.645  5.612
HELP LD
LD                                          Lists contents of a directory

    LD [pathname] [wild1 ... wild15] [options]

The LD command lets you list the contents of a directory 23 lines at
a time. The argument "pathname" specifies both the directory to be
listed and the first wildcard name. For example, "a>b>.list"
specifies entries in the directory A>B whose names match ".LIST". If
"pathname" is omitted, "          " is assumed; that is, all entries in the
current directory are selected.

The argument "wild1...15" specifies additional wildcard names. An
entry is selected if it matches either the "entryname" part of
pathname or one of the wildcard names.

If no entries are selected, the message "No entries selected" is displayed.

LD pauses after 23 lines of output, displays the prompt line
"--More--", and waits for your response. To see more entries, press
the carriage return. To suppress further output and return to
command level, type Q, Quit, N, or No. Any other response causes LD
to display the last header and to continue listing entries.

--More--          (user types Control-P)
QUIT.
OK 17:51:14  0.712  0.218  level 2
HELP COPY
COPY                                        Copies file system objects

     COPY source_pathname [target_pathname] [options]


source_pathname:  pathname of file system object to be copied

target_pathname:  pathname of output file system object
                  (if omitted, entryname portion
                  of "source_pathname" is used)
```

```
The COPY command copies file system objects (files, directories,
segment directories, access categories, and EPFs) from one directory
to another or within a directory. See the end of this file for
details on how EPFs (Executable Program Formats) are handled by the
COPY command.



--More--          (user types Control-P)
QUIT.
OK 17:51:40  0.509  0.087  level 3
RELEASE_LEVEL -TO 2
OK 17:51:44  0.096  0.000  level 2
START
(CR)

LD has the following options:

-NO_SORT, -NSORT                -CATEGORY_PROTECTED, -CATP
-SORT_DTM, -SORTM               -DEFAULT_PROTECTED, -DFLTP
-SORT_NAME, -SORTN              -SPECIFIC_PROTECTED, -SPECP
-SORT_DTB, -SORTB               -DETAIL, -DET
-SORT_SIZE, -SORTSZ             -BRIEF, -BR
-REVERSE, -RV                   -PROTECT, -PRO
-NO_WAIT, -NW                   -DTM
-SINGLE_COLUMN, -SGLCOL         -DTB
-NO_HEADER, -NHE                -SIZE
-WIDE                           -NO_COLUMN_HEADERS, -NCH
-FILE                           -ACAT
-SEGMENT_DIRECTORY, -SEGDIR     -DIRECTORY, -DIR

You can select options in any order.

The following screens give brief descriptions of each option.



--More--
```

In the above example, the user released the second invocation of HELP (which displayed information on the COPY command) so that a START command would continue the first invocation (which displayed information on the LD command).

The command environment does not treat static-mode programs and program EPFs in the same way after they terminate. Typing START after a static-mode program has terminated results in the continued execution of that program; typing START after an EPF has terminated results in the continued execution of the program that was suspended before the EPF was invoked.

Therefore, the RELEASE_LEVEL command, issued without options, is sometimes used to prevent the START command from restarting a static-mode program. For example:

```
OK 17:58:13  418.196  20.266
HELP LD
LD                                        Lists contents of a directory

    LD [pathname] [wildl ... wild15] [options]

The LD command lets you list the contents of a directory 23 lines at
a time. The argument "pathname" specifies both the directory to be
listed and the first wildcard name.  For example, "a>b>.list"
specifies entries in the directory A>B whose names match ".LIST". If
"pathname" is omitted, "          " is assumed;  that is, all entries in the
current directory are selected.

The argument "wildl...15" specifies additional wildcard names. An
entry is selected if it matches either the "entryname" part of
pathname or one of the wildcard names.

If no entries are selected, the message "No entries selected" is displayed.

LD pauses after 23 lines of output, displays the prompt line
"--More--", and waits for your response.  To see more entries, press
the carriage return.  To suppress further output and return to
command level, type Q, Quit, N, or No. Any other response causes LD
to display the last header and to continue listing entries.

--More--          (user types Control-P)
QUIT.
OK 17:58:37  0.706  0.100  level 2
ED
INPUT
THIS IS A TEST.
(CR)
EDIT
FILE A_TEST
OK 17:58:55  0.272  0.063  level 2+
START
INPUT
(CR)
EDIT
QUIT
OK 17:59:03  0.048  0.000  level 2+
RELEASE_LEVEL
Static mode program released. (rls)
OK 17:59:07  0.069  0.000  level 2
```

```
START
(CR)

LD has the following options:

-NO_SORT, -NSORT                    -CATEGORY_PROTECTED, -CATP
-SORT_DTM, -SORTM                   -DEFAULT_PROTECTED, -DFLTP


-SORT_NAME, -SORTN                  -SPECIFIC_PROTECTED, -SPECP
-SORT_DTB, -SORTB                   -DETAIL, -DET
-SORT_SIZE, -SORTSZ                 -BRIEF, -BR
-REVERSE, -RV                       -PROTECT, -PRO
-NO_WAIT, -NW                       -DTM
-SINGLE_COLUMN, -SGLCOL             -DTB
-NO_HEADER, -NHE                    -SIZE
-WIDE                               -NO_COLUMN_HEADERS, -NCH
-FILE                               -ACAT
-SEGMENT_DIRECTORY, -SEGDIR         -DIRECTORY, -DIR

You can select options in any order.

The following screens give brief descriptions of each option.



--More--
```

In the above example, the user attempted to use START to continue execution of the previously suspended HELP program. However, because ED is a static-mode program, ED was restarted instead, producing the INPUT prompt. Realizing the mistake, the user entered EDIT mode and used the QUIT subcommand of ED to exit the editor. Then, the user issued the RELEASE_LEVEL command without an option. When invoked without an option, RELEASE_LEVEL either releases the command level (if no static-mode program has been invoked at the current level) or releases the static-mode program invoked at the current level.

THE DUMP_STACK COMMAND

To display the contents of your stack, use the DUMP_STACK command. When you issue this command without specifying a particular stack frame as the starting point of the dump, DUMP_STACK searches the stack (starting with the most recently created stack frames) for a condition frame (a stack frame generated by the signaling of a condition). If it finds a condition frame, it starts the dump at that frame; otherwise, it starts the dump at the stack frame of its calling procedure (typically the internal procedure in STD$CP that executes internal commands).

When you are working with multiple command levels or with programs that invoke other programs or commands, DUMP_STACK is useful for examining the call history for a particular program.

By using the -ON_UNITS option of DUMP_STACK, as described earlier in this chapter, you can see what conditions are being caught by each invocation of a procedure whose stack frame is dumped. Some of the important command environment subroutines, such as STD$CP and the listener, are easily distinguished by the conditions they catch.

Although the list of conditions caught by any PRIMOS subroutine may change from revision to revision, you will find it useful to experiment with DUMP_STACK, Control-P, START, RELEASE_LEVEL, and related commands while you are not debugging a program. Later, when you are debugging a program, you will already know what the stack looks like and you will therefore be able to make more sense of the DUMP_STACK display.

THE INITIALIZE_COMMAND_ENVIRONMENT COMMAND

When you issue the INITIALIZE_COMMAND_ENVIRONMENT command (abbreviated ICE), your entire process state is reset to its original login state. PRIMOS does not do this by maintaining a copy of your login state; instead, it releases all resources in use by your process and performs much of the same initialization that it does when you log in. During program development, the ICE command is useful for tracking down program bugs and measuring performance by resetting your environment to a known state. For example, uninitialized variables may be more easily tracked down if you use the ICE command.

Before You Use ICE

Issuing the ICE command is similar to issuing the LOGOUT command, with one important difference: when you invoke ICE, no external logout program (supplied by your System Administrator) is run. Therefore, the ICE command has potentially damaging effects on certain Prime products and on some user-written products. For example, a system may use an external logout program that runs MPLUSCLUP to protect itself against a user logout in the midst of a MIDASPLUS transaction; this system cannot protect itself against a user issuing the ICE command at the same point in time.

In general, the danger involved with the ICE command is that one of its main features, the resetting of your environment even if the command processor stack or static-mode program stack is damaged, prevents the notification of subsystems that they are being terminated. A suspended subsystem that wishes to be notified when it is being terminated (that is, when it and its stack history are being released) makes an on-unit to catch the CLEANUP$ condition. Because information on conditions caught by a procedure resides in the stack frame for that procedure, and because ICE must avoid examining the stack in any way to prevent a damaged stack from aborting it, ICE does not signal the CLEANUP$ condition.

Therefore, use the ICE command only in circumstances where you are certain that it will cause no harm. In general, the safest way to use the ICE command is by issuing the following sequence of commands:

```
RELEASE_LEVEL -ALL
INITIALIZE_COMMAND_ENVIRONMENT
```

The RELEASE_LEVEL -ALL command does signal CLEANUP$ for all stack frames from the most recent to those for command level 1, notifying subsystems that they are being terminated, and it then releases those stack frames. If the command fails to complete successfully, then your stack is indeed damaged. In either case, it is then appropriate to issue the ICE command, because either you have successfully notified subsystems of impending termination, or your stack is damaged and they cannot be notified anyway.

THE REENTER COMMAND

You may use the REENTER command if you wish to reenter a program that catches the REENTER$ condition. The REENTER command currently ignores any command line arguments.

REENTER signals the condition REENTER$. The first program that catches the condition may:

- Continue the signal and return, in which case it is considered not reenterable (although a program invoked earlier may catch the condition)
- Return without continuing the signal, in which case it is considered not reenterable, and no other program can catch the condition
- Perform a non-local goto to one of its procedures, such as its command processor
- Set a flag in the information structure (pointed to by the condition frame header), and then return to restart the program, in which case REENTER functions like START. However, the program may distinguish START from REENTER by having the REENTER$ on-unit set a flag that is used elsewhere in the program

Using RELEASE_LEVEL With REENTER

As with the START command, you may wish to skip to a program so that you may use the REENTER command on that desired program rather than on a more recently invoked program that allows use of REENTER. You may use the RELEASE_LEVEL command to accomplish this.

In fact, in some cases, you may have to use RELEASE_LEVEL to reach the desired level so that REENTER works, even if programs between your current command level and the desired level do not intercept the REENTER$ signal. For example, if you invoke a program EPF that intercepts REENTER$ at command level 1, such as EMACS, and you Control-P to command level 2, invoke a static-mode program such as ED, then Control-P to command level 3, and invoke another static-mode program such as SLIST, you cannot use REENTER to reenter EMACS even though neither ED nor SLIST intercept REENTER$. Instead, PRIMOS displays the following message:

```
Attempt to proceed to non-executable program image. (listen_)
ER!
```

If this message is displayed, issue the RELEASE_LEVEL command to release the current command level. Then, attempt the REENTER command again. Continue alternating RELEASE_LEVEL and REENTER until you successfully reenter the desired program or you receive the message:

```
No subsystem supporting reentry exists on the stack. (ren)
ER!
```

The reason you cannot reenter EMACS on command level 1 with ED on level 2 and SLIST on level 3 is that, because ED and SLIST are both static-mode programs, SLIST may have overwritten ED. When REENTER$ is signaled, SLIST does not catch it, so it passes to the invocation of the listener at level 3. The listener's on-unit recognizes that the program at the level below it cannot proceed because it has been overwritten. The listener discontinues the signaling of REENTER$ and displays an error message; otherwise, if ED did catch REENTER$, it would be executing an overwritten program image. The listener cannot "skip over" ED and continue to signal the condition on the levels below it; therefore, the condition is not signaled for EMACS.

Programs That Handle REENTER

If a program does not catch the REENTER$ condition, you cannot use the REENTER command to continue execution of that program.

Only a few Prime-supplied programs (such as EMACS) catch the REENTER$ condition. User-written programs may also catch it.

Static-mode programs that allow START 1000 or START 1001 to perform some form of program restart may choose to switch to catching the REENTER$ condition instead, so that users may issue the REENTER command rather than having to remember START 1000 or START 1001. Such programs may continue to be static-mode after they are changed to catch the REENTER$ condition.

MINI-COMMAND LEVEL

Mini-command level is entered when a call to the listener is made when the user is already at the maximum command level allowed by the System Administrator or by the Project Administrator. Calls to the listener can occur when:

- The user types Control-P
- The program calls COMLV$
- The program encounters an error such as an access violation, illegal instruction, or insufficient memory

While at mini-command level, you can use only a small subset of internal PRIMOS commands. You cannot use external commands, and your abbreviations are temporarily disabled. These restrictions are in place to ensure that you can use certain critical internal commands without allowing you to use commands that may cause all resources to be exhausted. While at mini-command level, you should be able to determine why and how you got there.

When debugging a program, you may wish to use the DUMP_STACK command at mini-command level (along with COMOUTPUT to record the display on disk) to determine the stack history of your program leading up to the point where mini-command level was reached. For example, if you are at mini-command level because your program failed due to an access violation, the error message displayed by the on-unit for ACCESS_VIOLATION$ is followed by the information displayed when you enter mini-command level. (In fact, at Rev. 19.4, this information plus the prompt is 24 lines long, so the original error message may scroll off the top of your screen.) You use DUMP_STACK to trace the source of the ACCESS_VIOLATION$ error.

If possible, you should invoke the errant program starting at a lower command level, so that you avoid entering mini-command level when your program fails. This allows you more freedom in tracing the source of the error. For example, you might want to use BIND to display a map of the errant program EPF.

Other commands useful at mini-command level include LIST_EPF, RELEASE_LEVEL, INITIALIZE_COMMAND_ENVIRONMENT, START (if you reached mini-command level by typing Control-P), and LOGOUT.

WHAT CONTROL-P ACTUALLY DOES

When you type Control-P, PRIMOS signals the QUIT$ condition for your process. Any program may catch the QUIT$ condition and perform its own quit handling. Typically, however, this condition is caught by the default on-unit for the most recently created command level, because the listener establishes the default on-unit as the handler for any condition (the ANY$ condition).

The default on-unit performs the following actions when it catches the QUIT$ condition:

- Turns terminal output on (using an action similar to that performed by the COMOUTPUT -TTY command)
- Clears terminal input and output buffers (using an action similar to that performed by issuing the RSTERM command)
- Displays the QUIT. message on the terminal
- Suspends command input (using an action similar to that performed by issuing the COMINPUT -PAUSE command)
- Calls COMLV$ to invoke a new command level

The COMLV$ subroutine calls the listener.

When the user issues a START command with no arguments at a command level that has no static-mode program in use, the START command subroutine signals a condition that is caught by the listener. The on-unit invoked by the condition performs a nonlocal GOTO to a point in the listener that, after performing some internal cleanup of per-level information, simply returns to its caller; in this case, its caller is COMLV$. COMLV$ then returns to its caller, DF_UNIT_, which returns to the condition-signaling mechanism. Ultimately, the condition-signaling mechanism returns to the point at which the Control-P was typed in the original program.

If Your Program Catches QUIT$

Therefore, if you wish your program to catch the QUIT$ condition, you should ensure that it:

- Turns terminal output on (by calling COMO$$)
- Clears terminal input and output buffers (by calling TTY$RS)
- Displays a useful message on the terminal
- Suspends command input (by calling COMI$$)

- Calls COMLV$ to invoke a new command level, calls a command processor of its own, continues the signal, or resignals the condition, as appropriate

- Can proceed when a user types START (or the equivalent command in your program) in that any actions performed by the QUIT$ on-unit do not prevent program restart

The last requirement is often forgotten by programmers who write programs to catch QUIT$. If you write a QUIT$ on-unit, ensure that it either:

- Does nothing that prevents the program from continuing when the user types START (For example, closing one or more file units is not appropriate in a QUIT$ on-unit)

- Prevents a user from attempting to use START to restart the program (if your program must close file units when QUIT$ is signaled)

- Resets any conditions affected by the QUIT$ on-unit when the user types START so that the program may continue (such as reopening file units or, more typically, reinitializing a terminal display)

The last two options require your QUIT$ on-unit to maintain control after the user types START. In other words, your on-unit cannot simply continue the QUIT$ signal and return (which usually causes the signal to reach the default on-unit, DF_UNIT_). It must either call COMLV$ itself (after performing the actions listed above), in which case a START command returns control to the on-unit following the call to COMLV$, or it must resignal the QUIT$ condition, in which case a START command returns control to the on-unit following the call to SIGNL$.

Resignaling the QUIT$ Condition

Resignaling the QUIT$ condition is generally the best solution, although it is a bit tricky to implement. It combines two advantages:

- It allows the on-unit to regain control following a START.

- It does not require the on-unit to assume what action is desired by its invoking program upon receipt of a QUIT$ signal.

Typically used in a subroutine library, where the caller of the library wishes to catch QUIT$ in the caller's own way independent of the library, resignaling allows the designer of the subroutine library to deal with the QUIT$ condition in a way that does not interfere with the needs of a program that uses the subroutine library.

However, the trick to resignaling QUIT$ is that the on-unit doing the resignaling is invoked again for the second signal. To handle this,

you must use a flag in static or common storage that indicates whether the QUIT$ condition seen is signaled by the on-unit itself or by an external occurrence (such as a user typing Control-P) or by yet another subroutine package using this method).

For example, the following on-unit handles QUIT$ in a transparent manner:

```
quit_handler: proc(cp);

dcl cp ptr;  /* Pointer to the condition frame. */

dcl 1 cfh based(cp),
      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 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(12),
      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);

dcl my_signal bit(1) static init('0'b);

dcl cond_name char(32) var based;
```

```
dcl cnsig$ entry(fixed bin(15)),
    signl$ entry (char(32) var,ptr,fixed bin(15),ptr,
       fixed bin(15),bit(16) aligned),
    tnou entry(char(30),fixed bin(15)),
    break$ entry(fixed bin(15));

/* A second signal right at this point is not bad, because the
   second on-unit invocation then performs cleanup and sets
   MY_SIGNAL, causing the first on-unit invocation to simply
   continue the signal. */

call break$(1);  /* Inhibit quits. */

if my_signal  /* Did I signal this condition? */
   then do;  /* More precisely, have I already cleaned up? */
        call cnsig$(code);  /* Yes, just continue it. */
        call break$(0);  /* Enable quits. */
        return;  /* Continue the signal down the stack. */
        end;

/* Perform cleanup processing. */

    .
    .
    .

call tnou('Quit.',5);  /* Display a useful message. */

my_signal='1'b;  /* Set before enable to prevent multiple quits. */
call break$(0);  /* Enable quits. */
call signl$(cp->cond_name_ptr->cond_name,cp->ms_ptr,cp->ms_len,
     cp->info_ptr,cp->info_len,'E000'b4);
call break$(1);  /* Disable quits. */

call tnou('Restarting.',11);

/* Perform cleanup recovery code, or display "Not
   restartable" error message, or similar. */

    .
    .
    .

my_signal='0'b;  /* No longer signaling ourselves. */
call break$(0);  /* Enable quits. */

/* Quits after this point are acceptable because all cleanup
   has been reset by the cleanup recovery code. */

return;
end;  /* quit_handler: proc */
```

If the method shown above is used in a process-class library EPF, then a more reliable method of setting MY_SIGNAL to '0'b should be used. Simply initializing it when the linkage for the library is initialized is insufficient for a process-class library. You may wish to make it a common area (STATIC EXTERNAL) and have a standard initialization entrypoint in your library EPF initialize the bit to '0'b.

Be sure you reset MY_SIGNAL after calling SIGNL$ and BREAK$, but do not reset after calling CNSIG$ and BREAK$. Otherwise, a Control-P during the resignaling (and any processing it causes) would cause your on-unit repeat its cleanup activities, which may not be intended. The method shown above guarantees only one invocation of the cleanup code (shown as the first set of three periods) and only one corresponding invocation of the cleanup recovery code (the second set of three periods). It prevents reinvoking the cleanup code until the cleanup recovery code has been completely executed.

6

The Recursive Command Environment

The command environment is designed to accommodate recursive invocation of:

- The listener
- The command processor
- The iteration processor
- The command line reader
- The default on-unit
- User programs
- Many internal commands

Although user programs are part of this recursive environment, they often make use of resources that are not recursive (such as file units) or that are not dynamic (such as attach points).

WHAT IS A RECURSIVE RESOURCE?

A recursive resource is a resource that is identified separately by each invocation of a procedure. File units are not recursive, because if you invoke a program that opens a file on unit 2 and then calls another program that references unit 2, the second program references

the file opened by the first program; unit 2 is not a different file unit, even though it is being referenced by a different program.

WHAT IS A DYNAMIC RESOURCE?

A dynamic resource is one that, while not recursive, is acquired and released in a fashion that allows unique identification of one of its activations. For example, file units can be obtained dynamically. This facilitates their use by recursive subsystems; the programmer of such a subsystem must simply remember always to have the subsystem obtain its file units dynamically by using the k$getu key, as described in Volume II of this series.

WHAT IS A STATIC RESOURCE?

A static resource is one that is either always activated (and therefore is not actually acquired or released) or that can have only one activation at a time (in that a second activation of the resource destroys the first activation).

If file units are obtained statically, then it is possible for two programs to conflict in their static assignments of file unit numbers. Hence one of these programs cannot run after the other has been interrupted; or, if it can run, the second program cannot be continued after the first has finished.

THE CACHE ATTACH POINT AS A STATIC RESOURCE

An example of a static resource is the cache attach point, also described in Volume II of this series. There is only one cache attach point per process, yet many programs use it. A program may use the cache attach point explicitly (by calling AT$ subroutines or ATCH$$) or implicitly (by calling TSRC$$ or SRSFX$). Or a program may use the cache attach point because it has been invoked by the command processor as a command or by pathname. This is because the command processor attached to CMDNC0 using the cache directory and it accesses a pathname using SRSFX$.

Therefore, a program that uses the cache attach point may not always continue execution successfully if it is interrupted (by Control-P, for example) and if another program or command is invoked before the START command is issued to continue the original program. Even a mistyped command resets the cache attach point, because the command processor searches CMDNC0 for the command.

Even a program that invokes another program directly should do so without relying on the preservation of the cache attach point by the program being invoked.

These problems are not normally encountered when a user quits out of a program, invokes another program, then restarts the original program — particularly if the original program references the cache attach point only by calling PRIMOS subroutines such as TSRC$$ and SRSFX$. However, if the program happens to be in the midst of such a call when the interruption occurs — a rare but possible occurrence — the continuation of the program may produce strange results.

OTHER STATIC RESOURCES

Other static resources, when used by an EPF, may render that EPF static in nature, either with regard to all other EPFs or only to those EPFs that also use the resource.

File Names

File names in a directory are static. A program that uses a temporary file named TEMP.MYPROG in the user's home (or origin) directory cannot be recursively invoked. If a subroutine in a library EPF uses the temporary file, then two programs that use that subroutine cannot coexist in the same process.

If the file is in a system-wide location (such as CMDNC0) and has a static name (such as TEMP.MYPROG), then the problem becomes worse, because at that point two users running the same program simultaneously may encounter conflicts. Even if the user's home or origin directories — which at first glance seem to be per-process entities — are used, conflicts can result whenever two users have the same home or origin directories.

To make filenames dynamic, you must have your program or subroutine generate a unique filename. A reasonably unique filename is one that contains the date, the time of day (to ticks), the system name, and the usernumber of the user's process. Except for the system name, which is obtained by calling X$STAT, all of this information can be obtained by calling TIMDAT.

User's Display

The user's display screen is a static resource. If a subroutine is designed to display an important message at a particular spot on the screen, then multiple uses of that subroutine cause messages to be overwritten.

You must build a dynamic screen handler package to be used by programs and subroutines that wish to access the display screen in a dynamic fashion. These programs and subroutines would then call your package

to obtain a spot on the screen that suited their needs, and your package would find an appropriate spot based on the current state of the screen. If necessary, your package could overwrite existing messages only when approved by the user (in a fashion similar to the —More— prompt of the HELP command), or it could allow the user to display erased messages.

However, the dynamic screen handler package itself would not be dynamic; a second invocation of the package with a previous invocation already suspended would probably result in corruption of the display screen either immediately or when the previous invocation was continued.

Terminal Escape Sequences

Keep in mind that escape sequences sent to a terminal are static in nature. (An escape sequence consists of two or more characters that cause the terminal to perform a single action, such as clearing the screen or positioning the cursor.) If a subroutine sending such an escape sequence is interrupted in the midst of sending the sequence, (for example, via a forced logout (signaling LOGOUT$) or via Control-P), the on-unit for the interruption must not send any characters to the terminal. Otherwise, the first few characters it sends will be interpreted by the terminal as being part of the interrupted escape sequence. Such a window is rarely encountered; but it can happen, and it corrupts the screen.

If on-units for asynchronous conditions (such as LOGOUT$, QUIT$, PH_LOGO$, and so on) do not write any messages to the terminal either directly or indirectly (by calling other subroutines), the problem is avoided. Instead, the on-units might record the desired message in memory and set a flag to be picked up by the mainline code at a point not in the midst of an escape sequence.

INDEX

Index

Symbols

; (command separator character), 2-3

~ (tilde), 2-2, 4-11

A

ABBREV command, 2-3

Abbreviation processor, 1-19

Abbreviations,
  disabled at mini-command level, 5-14

-ACCESS_CATEGORY bit, 3-21, 4-33

ALC$RA subroutine, 3-9, 3-10, 3-13

Allocating,
  linkage areas, via EPF$ALLC, 4-34

ALS$RA subroutine, 3-9, 3-10, 3-12

ANY$ condition, 1-22, 5-15

Applications,
  command environment support for, 1-7
  defined, 1-7

Attach point,
  cache, 6-2

B

BIND,
  NO_GENERATION subcommand, 2-2
  NO_ITERATION subcommand, 2-2
  NO_TREEWALK subcommand, 2-2
  NO_WILDCARD subcommand, 2-2

-BOTTOM_UP bit, 3-21

C

Cache attach point,
  as a static resource, 6-2

Calling sequences,
  command, detailed, 3-15, 3-17
  complete, 3-26, 3-29
  error codes, 3-5
  for command functions, 3-6
  for commands, 3-3, 3-4
  for program EPFs, 3-1
  for programs, 3-3

Closing,
  file after EPF$RUN returns, 4-25

code argument,
  for CP$, 4-12
  for EPF$MAP subroutine, 4-29
  for EPF$RUN, 4-22

COMINPUT files,
  command environment support for, 1-3

Command,
  defined, 1-8
  invocation, 1-10
  name, determination of, 2-4

Command calling sequence, 3-3
  arguments for, 3-3
  error codes for, 3-5

Command environment, 1-1
  (See also command processing information)
  abbreviation processor, 1-19
  command features decoder, 1-21
  command interface, 1-10
  command line reader, 1-19
  command preprocessor, 1-21
  command processor, 1-20
  command prompter, 1-19
  default on-unit, 1-22
  features for applications, 1-7
  features for COMINPUT files, 1-3
  features for CPL programs, 1-4
  features for interactive users, 1-2
  features for user-written functions, 1-7
  features for user-written programs, 1-6
  key modules, 1-16

Command environment (continued)
  listener, 1-17
  program invokers, 1-22

Command features decoder, 1-21

Command function calling sequence, 3-6, 3-8

Command function invocation,
  via CP$, 4-13
  via EPF$INVK, 4-27
  via EPF$RUN, 4-18

Command functions, 4-4
  actions of, 3-6
  arguments for calling sequence, 3-7
  behavior when invoked as commands, 4-4
  needing command name, 3-27
  needing local CPL variables, 3-27
  sample programs, 3-11
  special cases of, 3-26
  usable as commands, 3-27

Command information structure,
  two versions of, 4-23
  use of with EPF$RUN, 4-24

Command interface, 1-10
  for one program invoking another, 1-11
  levels of complexity, 1-10

Command invocation, (See also command processing information)
  calling sequence, 3-2 to 3-4
  command line, 1-12
  defined, 1-11
  error codes for, 3-5
  limits on, 1-13
  severity code, 1-12

Command levels, 1-17, 5-2
  defined, 1-17
  listener, 1-17
  listener, the, 5-2
  mini-command level, 1-18, 5-14
  multiple, 5-3
  releasing, 5-5, 5-6

Command line,
  accepted by EPF, 1-12
  as argument in calling
    sequence, 3-3
  as argument to CP$, 4-11
  use of tilde (~) in front of,
    4-11

Command line reader, 1-19
  recursive invocation of, 6-1

Command names, determined by
    command processor, 2-4

Command preprocessor, 1-21

Command processing information,
    1-13, 3-15, 3-16, 3-18
  -ACAT bit, 3-21, 4-33
  -BOTTOM_UP bit, 3-21
  command name, 3-19
  CPL local variables pointer,
    3-20
  -DIRECTORY bit, 3-21, 4-33
  -FILE bit, 3-21, 4-33
  iteration bit, 3-22
  -RBF bit, 3-21, 4-33
  sample program, 3-22, 4-51
  -SEGMENT_DIRECTORY bit, 3-21,
    4-33
  treewalk bit, 3-22, 4-31
  -VERIFY bit, 3-21, 4-33
  version, 3-19
  -WALK_FROM bit, 3-22
  -WALK_TO bit, 3-22
  wildcard bit, 3-22, 4-31

Command processor, 1-20, 2-1
  ABBREV command, handling of,
    2-3
  actions when invoked by CP$,
    4-7
  calls STD$CP, 1-20
  command separator character
    (;), handling of, 2-3
  determines command name, 2-4
  determines command type, 2-5
  evaluates function references,
    2-4
  evaluates variable references,
    2-4
  expression evaluator, 1-20
  inhibition of features, 2-2,
    2-5

Command processor (continued)
  interface with commands, 1-10,
    1-11
  invocation modules, 2-7
  invokes commands, 2-7
  iteration, handling of, 2-5
  listener, 1-17
  listener, the, 5-2
  name generation, handling of,
    2-7
  -NO_VERIFY, handling of, 2-7
  recursive invocation of, 6-1
  removes null tokens, 2-4
  RESUME command, 2-5
  sequence of actions, 2-1
  simple iteration, handling of,
    2-5
  stack, 5-1
  treewalking, handling of, 2-6
  -VERIFY, handling of, 2-7
  wildcards, handling of, 2-6

Command processor stack, 5-1
  viewed with DUMP_STACK, 5-2

Command prompter, 1-19

Command separator character, 2-3

command-information argument,
  for EPF$RUN subroutine, 4-23

command-line argument,
  for EPF$RUN subroutine, 4-22

Commands,
  DUMP_STACK, 5-10
  external, 1-10
  format of, 4-3
  ICE, 1-16, 5-11
  INITIALIZE_COMMAND_ENVIRONMENT,
    1-6, 5-11
  interface with command
    processor, 1-11
  internal, 1-9, 4-3
  RDY, 5-4
  recursive invocation of, 6-1
  REENTER, 5-12
  RELEASE_LEVEL, 5-5
  REN, 5-12
  resident in CMDNC0, 4-2
  resident within PRIMOS, 4-2
  RLS, 5-5

Commands (continued)
  START, 5-14
  usable as command functions, 3-27

Common storage,
  releasing, 5-6

Complete calling sequence, 3-26, 3-29

Conditions,
  ANY$, 1-22
  LINKAGE_ERROR$, 1-14
  NO_AVAIL_SEGS$, 1-16
  PAGING_DEVICE_FULL$, 1-16
  QUIT$, 5-15
  REENTER$, 5-12
  STORAGE, 1-15
  SYSTEM_STORAGE$, 1-15

CONTROL-P (Quit), 5-15
  (See also QUIT$ condition)

CP$ subroutine, 1-20, 3-1, 4-9
  actions of, 4-7
  calling sequence, 4-10
  command-line argument of, 4-11, 4-14
  cpl-local-vars-ptr, 4-13, 4-15
  error codes returned by, 4-17
  error-code argument of, 4-12, 4-14
  flags argument of, 4-12
  ftn-fcn-ptr argument of, 4-14
  function-call bit, 4-12, 4-14
  inhibit-evaluation bit, 4-12, 4-15
  rtn-fcn-ptr argument, 4-13
  severity-code argument of, 4-12, 4-14
  used for command invocation, 4-9
  used for function invocation, 4-13
  used for program invocation, 4-9
  used for recursive invocation, 4-54
  when to use it, 4-6

CPL programs,
  abilities of, 4-4
  command environment support for, 1-4

CPL variables,
  pointed to by cpl-local-vars-ptr, 4-15
  used by command functions, 3-27

cpl-local-vars-ptr, 4-15
  argument to CP$, 4-13

CPL-program invoker, 2-7

D

Decoder, command features, 1-21

Default on-unit, 1-22
  actions on catching QUIT$, 5-15
  recursive invocation of, 6-1

Detailed command calling sequence, 3-15, 3-17

DF_UNIT_ (See Default on-unit)

-DIRECTORY bit, 3-21, 4-33

DUMP_STACK command, 5-10
  -ON_UNITS option, 5-2, 5-11
  to display call history of a program, 5-10
  use at mini-command level, 5-14
  used to track program errors, 1-18
  used to view command processor stack, 5-2
  used to view your stack, 5-10

Dynamic resources, 6-2

Dynamically allocated storage,
  releasing, 5-6

E

EPF,
  invocation by CP$ subroutine, 4-9
  invocation by EPF$INVK subroutine, 4-27
  invocation by EPF$RUN subroutine, 4-20
  most flexible format for programming instructions, 4-4
  program, calling sequence, 3-1
  recursive invocation of, 4-54

EPF calling sequence,
  arguments for, 3-1
  command sequence, 3-2
  program sequence, 3-2

EPF generation and use,
  phase 10 (removal from memory), 4-37
  phase 5 (linkage allocation), 4-34
  phase 6 (linkage initialization), 4-34
  phase 7 (entrypoint invocation), 4-37
  sample program, 4-47

EPF id, 4-24

EPF invoker, 2-7

EPF$ALLC subroutine, 4-34
  calling sequence, 4-35
  error codes, 4-42

EPF$CPF subroutine, 4-31
  calling sequence, 4-32
  epf-info structure, 4-31
  error codes, 4-42
  sample program using, 4-51
  wildcard bit, 4-31

EPF$DEL subroutine, 4-37
  calling sequence, 4-39
  error codes, 4-44

EPF$INIT subroutine, 4-34
  calling sequence, 4-36
  error codes, 4-43

EPF$INVK subroutine, 3-1, 3-2
  calling, 4-37
  calling sequence, 4-38
  compared with EPF$RUN, 4-8
  error codes, 4-44
  invoking EPF$ALLC before using, 4-34
  invoking EPF$CPF before using, 4-31
  invoking EPF$DEL after using, 4-37
  invoking EPF$INIT before using, 4-34
  invoking EPF$MAP for, 4-29
  key argument, 4-34
  opening file for, 4-28
  steps in using, 4-27
  used for recursive invocation, 4-54
  when to use it, 4-8

EPF$MAP subroutine, 4-29
  access argument, 4-29
  calling sequence, 4-30
  code argument, 4-29
  key argument, 4-29
  unit argument, 4-29

EPF$MAP subroutines,
  error codes, 4-40

EPF$RUN subroutine, 3-1, 4-18
  actions of, 4-8
  calling sequence of, 4-21
  checking returned code value, 4-25
  checking returned command status, 4-25
  command-information structure, 4-23
  command-line argument, 4-22
  EPF id, 4-24
  error codes returned by, 4-26
  error-code argument, 4-22
  file-unit argument, 4-22
  function-call bit, 4-24
  invoking, 4-20
  key argument, 4-20
  opening EPF file before calling, 4-19
  rtn-fcn-ptr, 4-24
  severity-code argument, 4-22
  steps in using, 4-18

EPF$RUN subroutine (continued)
used for recursive invocation, 4-54
using and freeing returned value structure, 4-25
when to use it, 4-8

Epf-info structure, 4-31

Error codes,
checking code returned by EPF$RUN, 4-25

E$BARG, 3-5, 4-18, 4-43
E$BKEY, 4-26, 4-41, 4-43
E$BNAM, 3-5, 4-18
E$BPAR, 3-5, 4-41 to 4-44
E$BUNT, 4-26, 4-41
E$BVER, 4-18, 4-27, 4-41 to 4-44
E$CMND, 3-5, 4-18
E$DIRE, 4-17
E$ECEB, 1-14, 4-44
E$EOF, 4-17, 4-26
E$EPFL, 4-41
E$EPFT, 4-41 to 4-45
E$FIUS, 4-17
E$FNTF, 4-18
E$ILTD, 4-42, 4-43
E$ILTE, 4-43
E$ITRE, 3-5, 4-18
E$IVCM, 3-6
E$MISA, 3-6
E$NDAM, 4-18, 4-40, 4-41
E$NINF, 4-18
E$NMLG, 3-5
E$NMTS, 4-26, 4-41
E$NMVS, 4-27, 4-41
E$NOVA, 4-41
E$NRIT, 4-17, 4-40
E$ROOM, 4-26
E$SWPR, 4-45
E$UNOP, 4-26, 4-40
returned by EPF$ subroutines, 4-40
returned by EPF$ALLC, 4-42
returned by EPF$CPF, 4-42
returned by EPF$DEL, 4-44
returned by EPF$INIT, 4-43
returned by EPF$INVK, 4-44
returned by EPF$MAP, 4-40
returned by EPF$RUN, 4-26

error-code argument,
of EPF$RUN subroutine, 4-22

Escape sequences,
as a static resource, 6-4

Evaluation of function and variable references, 2-4

Expression evaluator, 1-20

External commands, 1-10

F

File,
closing after EPF$RUN returns, 4-25
opening for VMFA access, 4-19

-FILE bit, 3-21, 4-33

File names,
as a static resource, 6-3
creating dynamic file names, 6-3
search order of, 1-10

File unit number,
as argument to EPF$MAP, 4-29

File units,
as a static resource, 6-1

file-unit argument,
of EPF$RUN subroutine, 4-22

FRE$RA subroutine, 4-45
calling sequence, 4-46
when to use it, 4-8

Freeing memory,
via FRE$RA subroutine, 4-45
via ICE command, 1-16

Function invocation, (See also Command function invocation)
command line, 1-12
defined, 1-11
returned character string, 1-12
severity code, 1-12

Function invocation (continued)
via EPF$INVK, 4-27
via EPF$RUN, 4-18

Function references, evaluation of, 2-4

function-call argument, 3-9

function-call bit, 4-12, 4-14, 4-24

Functions, (See also Command functions)
command environment support for, 1-7
defined, 1-9, 1-12
interaction with command processor, 1-11
invocation of, 1-11

I

ICE command, 5-11
use of, 1-16

Inhibit-evaluation bit, 4-12, 4-15

Inhibition of command processor features, 2-5

INITIALIZE_COMMAND_ENVIRONMENT command, 5-11
use of, 1-16

Initializing,
linkage areas, via EPF$INIT, 4-34

Interactive users,
command environment support for, 1-2

Internal commands, 1-9

Internal-command invoker, 2-7

Invocation, (See also Command invocation; Function invocation; Program invocation)
limits on, 1-13
of commands, 1-11
of commands, by command processor, 2-7
of functions, 1-11
of programs, 1-10
of programs, from within programs, 4-1
recursive, 4-54

Invoking an EPF,
sample program, 4-47

Iteration,
handling of by command processor, 2-5
simple, 2-5

Iteration bit, 3-22

Iteration processor,
recursive invocation of, 6-1

K

k$getu key, 4-19

k$invk key, 4-20

k$invk_del key, 4-20

k$restore_only key, 4-20

k$vmr key, 4-19

key argument,
for EPF$INVK subroutine, 4-34
for EPF$MAP subroutine, 4-29
for EPF$RUN subroutine, 4-20

L

LB$SET subroutine, 3-27

Linkage areas,
releasing, 5-6

LINKAGE_ERROR$ condition, 1-14, 1-15

LIST_LIMITS,
  use of, 1-14

LIST_SEGMENTS,
  use of, 1-14

Listener, 1-17

Listener, the, 5-2
  and mini-command level, 5-14
  multiple invocations, 5-3
  recursive invocation of, 6-1

Long prompt, 5-5

LV$GET subroutine, 3-27

M

Memory,
  releasing via FRE$RA subroutine, 4-45
  system-wide limits on, 1-15

Mini-command level, 1-18, 5-14

N

Name generation,
  handled by command processor, 2-7

Names of commands, determined by command processor, 2-4

NO_AVAIL_SEGS$ condition, 1-16

-NO_VERIFY option,
  handled by command processor, 2-7

Null tokens, removal of from command line, 2-4

NW$ filename prefix, 4-4

NX$ filename prefix, 4-4

O

Opening,
  EPF file for VMFA access, 4-19
  file for VMFA read, possible error codes, 4-40

P

PAGING_DEVICE_FULL$ condition, 1-16

Pointer, returned value, 3-9

Procedure code storage,
  releasing, 5-6

Program EPF, calling sequence, 3-1

Program invocation,
  calling sequence, 3-2, 3-3
  deciding which interface to use, 4-6
  defined, 1-10
  from within programs, 4-1
  limits on, 1-13

Programs,
  command environment support for, 1-6
  format of, 4-3
  interface with command processor, 1-10
  invoking programs from, 4-1
  resident on disk, 4-2

Prompter, command, 1-19

Prompts, set by RDY command, 5-5

Q

QUIT$ condition, 5-15
  as handled by default on-unit, 5-15

QUIT$ condition (continued)
how your program can handle it, 5-15, 5-16
resignaling the condition, 5-16
sample program, 5-16

R

-RBF bit, 3-21, 4-33

RDY command, 5-4
in LOGIN.CPL files, 5-5
to specify system prompts, 5-4

Reader, command line, 1-19

Recursive command environment, 6-1
creating dynamic screen handlers for, 6-3
file units not recursive, 6-1
generating dynamic file names for, 6-3
handling terminal escape sequences in, 6-4
limits on use of cache attach point, 6-2

Recursive invocation of EPFs, 4-54
behavior of static storage during, 4-54
redirecting terminal I/O during, 4-55

REENTER command, 5-12
used with RELEASE_LEVEL command, 5-13

REENTER$ condition, 5-12

RELEASE_LEVEL command, 5-5, 5-7
releasing to a particular level, 5-7
resources released by, 5-6
used to restart a suspended program, 5-7
used with REENTER command, 5-13

Releasing,
memory holding returned value, 4-45
resources, 5-6

Removing EPF from memory,
via EPF$DEL subroutine, 4-37

REN command (See REENTER command)

Resources,
dynamic, 6-2
per-user limits, 1-13
releasing, 5-6
static, 6-2
system-wide limits, 1-16

Restarting suspended programs, 5-7
with REENTER command, 5-13

RESUME command,
special treatment by command processor, 2-5

Returned character strings, 1-12

Returned command status,
checking after EPF$RUN, 4-25

Returned function value pointer (See rtn-fcn-ptr)

Returned function value structure, 4-15
accessed from FORTRAN, 4-15, 4-16
accessed from PL1G, 4-15
deallocating memory via FRE$RA, 4-45
using and freeing it after calling EPF$RUN, 4-25

Returned value,
defined, 3-6
freeing memory used by, 4-45

Returned value pointer (See rtn-fcn-ptr)

RLS command (See RELEASE_LEVEL command)

RLS command (RELEASE_LEVEL), 5-5

rtn-fcn-ptr, 3-9, 4-13, 4-14
  declaration of structure, 4-15
  for EPF$RUN subroutine, 4-24

S

Sample programs,
  command functions, 3-11, 3-14
  handling command processing information, 4-51
  handling QUIT$ condition, 5-16
  showing EPF invocation and execution, 4-47
  using command processing information, 3-22

Screen handlers, building, 6-3

Search order for filenames, 1-10

Segment access,
  as argument to EPF$MAP subroutine, 4-29

-SEGMENT_DIRECTORY bit, 3-21, 4-33

Severity code,
  as argument to CP$, 4-12
  as argument to EPF$RUN, 4-22
  for command calling sequence, 3-6
  returned by EPF, 1-12

Simple program,
  defined, 1-8

SRCH$$ subroutine,
  used to open file for VMFA read, 4-19

SRSFX$ subroutine,
  used to open file for VMFA read, 4-19

Stack frames,
  releasing, 5-6

Stack, command processor, 5-1

Standard command processor, 1-20
  (See also Command processor)

START command, 5-14

Static resources, 6-2
  cache attach point, 6-2
  escape sequences sent to terminal, 6-4
  file names, 6-3
  user's display screen, 6-3

Static storage,
  and recusive invocation, 4-54

Static-mode programs,
  limits on flexibility of, 4-4

Static-mode-program invoker, 2-7

STDCP$ subroutine, 1-20

STORAGE condition, 1-15

Storage, static, 4-54

Suffixes,
  search order of, 1-10, 4-3

Suspended programs,
  restarting, 5-7

System prompts, 5-5

SYSTEM_STORAGE$ condition, 1-15

T

Terminal I/O,
  redirection during recursive invocation of EPFs, 4-55

Tilde, use of, 2-2

Treewalk bit, 3-22, 4-31

Treewalking,
  handled by command processor, 2-6
  in command processing information, 3-22
  in epf-info structure, 4-31

Treewalking (continued)
  options for, 2-6
  specified in command
    information structure, 4-23
  specified in epf-info
    structure, 4-31

TSRC$$ subroutine,
  used to open file for VMFA
    read, 4-19

U

User programs,
  recursive invocation of, 6-1

User-written functions,
  command environment support
    for, 1-7

User-written programs,
  command environment support
    for, 1-6

V

Variable references, evaluation
    of, 2-4

-VERIFY bit, 3-21

-VERIFY option,
  handled by command processor,
    2-7

W

-WALK_FROM bit, 3-22

-WALK_TO bit, 3-22

Wildcard bit, 3-22, 4-31

Wildcards,
  handled by command processor,
    2-6
  in command processing
    information, 3-21
  in epf-info structure, 4-31
  options for, 2-6

SURVEY

READER RESPONSE FORM

DOC10057-1LA Advanced Programmer's Guide, Volume III

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

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

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

READER RESPONSE FORM

DOC10057-1LA Advanced Programmer's Guide, Volume III

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

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

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

READER RESPONSE FORM

DOC10057-1LA Advanced Programmer's Guide, Volume III

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

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

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

READER RESPONSE FORM

DOC10057-1LA Advanced Programmer's Guide, Volume III

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

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

___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

Readability: ___hard to understand ___average ___very clear

Technical level: ___too simple ___about right ___too technical

Technical accuracy: ___poor ___average ___very good

Examples: ___too many ___about right ___too few

Illustrations: ___too many ___about right ___too few

3. What features did you find most useful? _____

_____

_____

_____

4. What faults or errors gave you problems? _____

_____

_____

_____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

DOC10057-1LA