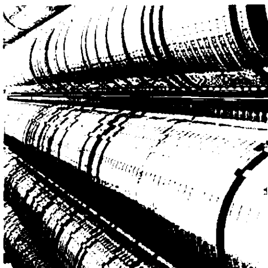
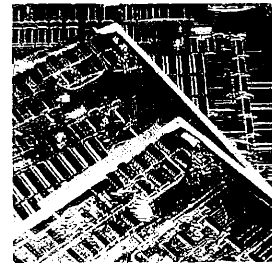
Prime Computer, Inc.

DOC8691-1LA

Programmer's Guide to BIND and EPFs

Revision 19.4





Programmer's Guide to BIND and EPFs

First Edition

by

James Burley, Evelyn Burns,
Jacki Forbes, Sarah Lamb,
Alice Landy

Updated for Rev. 22.0

by

William T. Carbonneau

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

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

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

Copyright © 1988 by Prime Computer, Inc. All rights reserved.

PRIME, PR1ME, PRIMOS, and the PRIME logo are registered trademarks of Prime Computer, Inc. DISCOVER, INFO/BASIC, INFORM, MIDAS, MIDASPLUS, PERFORM, Prime INFORMATION, PRIME/SNA, PRIMELINK, PRIMENET, PRIMEWAY, PRIMIX, PRISAM, PST 100, PT25, PT45, PT65, PT200, PW153, PW200, PW250, RINGNET, SIMPLE, 50 Series, 400, 750, 850, 2250, 2350, 2450, 2550, 2655, 2755, 4050, 4150, 6350, 6550, 9650, 9655, 9750, 9755, 9950, 9955, and 9955II are trademarks of Prime Computer, Inc.

PRINTING HISTORY

First Edition (DOC8691-1LA) January 1986 for Revision 19.4
Update 1 (UPD8691-11A) October 1988 for Revision 22.0

CREDITS

Editorial: Thelma Henner
Project Support: Susan Miano
Illustration: Mingling Chang
Document Preparation: Mary Mixon
Production: Judy Gordon

HOW TO ORDER TECHNICAL DOCUMENTS

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

United States Customers

Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).

International

Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT

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

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

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

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

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

Contents

ABOUT THIS BOOK xi

PART I -- CREATING AND RUNNING EPFS

1 OVERVIEW OF BIND AND EPFS

Prime's Linkers and Loaders 1-1
Linking Versus Loading 1-2
Addressing Modes 1-2
The LOAD Linking Loader 1-3
The SEG Linking Loader 1-3
The BIND Linker 1-4
Types of Runfiles 1-6
Static Runfiles 1-6
Dynamic Runfiles 1-6
Types of EPFs 1-7
The Stages of a Program EPF 1-8
EPFs and Memory Allocation 1-9
Sharing Program EPFs 1-10
EPFs and DBG 1-10
Remote EPFs 1-10

2 WORKING WITH BIND

Before You Begin... 2-2
Naming Files 2-2
How to Use BIND 2-5
Linking a Program With One Command 2-6
Running BIND Interactively 2-7
Basic Subcommands for Linking With BIND 2-8
Essential Subcommands 2-8
Other Useful Subcommands 2-8
After Your EPF Is Created 2-8
Using the LOAD Subcommand 2-9
Using the LIBRARY Subcommand 2-9
Using the FILE Subcommand 2-12
Using the QUIT Subcommand 2-13
Using the MAP -UNDEFINED Subcommand 2-15
Using the HELP Subcommand 2-16

Examples of a Standard Linking Sequence Using BIND 2-17
Retrying a Linking Sequence 2-18
Reloading a Module 2-19

3 RUNNING EPFS

Running Your Program 3-1
Sample Sessions 3-2

PART II -- USING BIND AND EPFS

4 PROGRAMMING WITH EPFS

Overview 4-1
Programming With EPFs Versus Static-mode Programs 4-2
Guidelines for Programming With EPFs 4-2
Calling Programs 4-4
What Programs Can EPFs Call? 4-4
What Programs Can CPL Programs Call? 4-5
What Programs Can Static-mode Programs Call? 4-5
Limitations Involved With Static-mode Programs 4-5

5 CONVERTING TO EPFS

Why Do I Want to Convert to EPFs? 5-1
EPFs and PRIMOS 5-2
Suspending EPFs 5-3
EPFs Calling EPFs 5-3
Building EPFs With BIND 5-3
EPFs and the Command Processor Stack 5-3
Conversion Rules 5-3
What Can't I Convert? 5-4
FTN Programs Using the -PBECB Option 5-4
Routines With Modifiable Data in the Program 5-4
Use of CALL EXIT as a Pause Function 5-5
Programs Sharing Linkage Segments 5-5
Failure to Declare Memory Used 5-6
Programs That Call R-mode or S-mode Subroutines 5-7
Requiring Defined Initial Values for Registers 5-7

Programs That Depend on Automatic Initialization 5-7
Conversion Procedures 5-8
Rewriting Linking Sequences 5-8

6 LIBRARIES AND LIBRARY EPFS

What Is a Library? 6-1
When Is a Library Useful? 6-1
How to Use a Library 6-2
Types of Libraries 6-3
Nonshared Libraries 6-3
Static Shared Libraries 6-3
Library EPFs 6-4
Linking a Program to a Library Routine By Name 6-4
How Dynamic Linking Works 6-4
Library EPFs 6-5
Program-class Libraries 6-5
Process-class Libraries 6-6
Interaction of the Classes of Library 6-6
How to Create Your Own Library EPFs 6-7
How to Use a Library EPF 6-9
Creating an Entrypoint Search List File 6-9
Enabling an Entrypoint Search List File 6-10
Disabling an Entrypoint Search List File 6-11
Permanently Enabling Your Entrypoint Search List File 6-11
Use of Private Entrypoint Search Lists With Phantoms or Batch Jobs 6-11
Using the DYNT Subcommand of BIND 6-12

7 TROUBLESHOOTING

Problems You May Run Into 7-1
Running Out of Individual User Resources 7-1
Command Environment Depth 7-2
Command Environment Breadth 7-2
Segments 7-2
Mini-command Level 7-3
Running Out of System Resources 7-7
Problems With In-use EPFs 7-7
Problems With BIND 7-8
Problems With Libraries and Search Rules 7-8
Using DBG 7-10

PART III -- REFERENCE

8 BIND SUBCOMMANDS DICTIONARY

BIND 8-3
Linking Commands 8-4
ALLOCATE 8-4
CHANGE_SYMBOL_NAME 8-4
COMMENT 8-5
COMMON_WARNING 8-5
COMPRESS 8-5
DYNT 8-6
ENTRYNAME 8-6
FILE 8-7
HELP 8-8
INITIALIZE_DATA 8-8
LIBMODE 8-8
LIBRARY 8-9
LOAD 8-9
MAIN 8-11
MAP 8-11
NO_COMMON_WARNING 8-15
QUIT 8-15
RELOAD 8-15
RESOLVE_DEFERRED_COMMON 8-16
SEARCH_RULE_VERIFY 8-16
SYMBOL 8-17
VERSION 8-18
Executable Program Files as Commands 8-18
ITERATION 8-19
NAMEGENPOS 8-19
NO_GENERATION 8-19
NO_ITERATION 8-19
NO_TREEWALK 8-20
NO_WILDCARD 8-20
TREEWALK 8-20
WILDCARD 8-20

9 EPF COMMANDS DICTIONARY

Overview 9-1
Choosing Which Command to Use 9-2
EPF Mini-commands 9-3
EPF-related PRIMOS Commands 9-4
EXPAND_SEARCH_RULES 9-5a
INITIALIZE_COMMAND_ENVIRONMENT 9-6
LIST_EPF 9-8
LIST_LIBRARY_ENTRIES 9-19
LIST_LIMITS 9-21
LIST_MINI_COMANDS 9-22
LIST_SEARCH_RULES 9-23

LIST_SEGMENT 9-25
REMOVE_EPF 9-29
SET_SEARCH_RULES 9-32
Using the COPY Command
With EPFs 9-36
Using COPY to Replace an
Open EPF File 9-36
REPLACE (.RPn) Files 9-37

10 EPFS CALLING PROGRAMS

Understanding Commands, Command
Functions, and Programs 10-2
Internal PRIMOS Commands 10-2
Internal CPL Command Functions 10-2
External Commands 10-3
Users' Programs 10-3
CP$ EPF$RUN, and FRE$RA 10-3
Passing Information to and From
Programs 10-4
Command Lines 10-4
Severity Codes 10-5
Returned Text Strings 10-7
Understanding Program Invocation 10-7
Use of Shared Memory 10-10
Limits on Program Invocation 10-10
The Command Interface 10-10
Using the CP$ Subroutine 10-12
Using CP$ to Invoke an
Internal PRIMOS Command 10-12
Using CP$ to Invoke a Command
or a Program 10-13
Using CP$ to Invoke a Function 10-14
Command Preprocessing 10-17
Terminal Input and Output 10-19
Error Codes From CP$ 10-19
If a Program Invokes Itself 10-21
Sample Program 10-25

APPENDIXES

A BIND ERROR MESSAGES A-1

B EPF ERROR MESSAGES B-1

C GLOSSARY C-1

INDEX X-1

About This Book

This book is an introduction to BIND and EPFs (Executable Program Formats). The book is divided into three parts and a set of appendixes:

Part I: Creating and Running EPFs

- Chapter 1 provides the general background information you need about BIND and EPFs
- Chapter 2 helps you start to use fundamental commands with BIND in order to create EPFs.
- Chapter 3 shows you how to run an EPF, using the RESUME command.

Part II: Using BIND and EPFs

- Chapter 4 discusses the programming restrictions and limitations you must be aware of when you create EPFs.
- Chapter 5 shows you how to convert your static-mode programs into EPFs.
- Chapter 6 discusses libraries and how to build an EPF library.
- Chapter 7 discusses some problems you may encounter, and ways to resolve them.

Part III: Reference

- Chapter 8 is a dictionary of BIND subcommands.
- Chapter 9 is a dictionary of EPF-related PRIMOS commands.
- Chapter 10 explains how EPFs call other programs.
- The appendixes list BIND error messages and EPF error messages, and provide a glossary.

This book assumes that you have some programming background in a high-level language.

OTHER USEFUL BOOKS

Other useful books for users of BIND and EPFs include the following:

- PRIMOS User's Guide, DOC4130-5LA, which provides an overview of programming in a high-level language on a Prime Computer.
- PRIMOS Commands Reference Guide, DOC3108-7LA, which contains information on format and usage of all PRIMOS user commands.
- The reference guides for the programming languages you use on your Prime system.

In addition, the following books are valuable if you do systems-level programming:

- The Advanced Programmer's Guide, Volume 0: Introduction and Error Codes, DOC10066-3LA
- The Advanced Programmer's Guide, Volume I: BIND and EPFs, DOC10055-1LA
- The Advanced Programmer's Guide, Volume II: File System, DOC10056-2LA
- The Advanced Programmer's Guide, Volume III: Command Environment, DOC10057-1LA

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 letters indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase letters.	COPY
lowercase	In command formats, words in lowercase letters indicate items for which the user must substitute a suitable value.	RESUME pathname
Abbreviations	If a command or statement has an abbreviation, it is indicated by underlining. In cases where the command or the directive itself contains an underscore, the abbreviation is shown below the full name, and the name and abbreviation are placed within braces.	COMOUTPUT { LIST_EPF / LE }
Underlining in examples	In examples, user input is underlined, but system prompts and output are not.	OK, RESUME MYPROG This is the output of MY_PROG.CPL OK,
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or two or more items.	LD [-DETAIL / -SIZE]
Braces { }	Braces enclose a list of items. Choose only one item.	CLOSE { filename / -ALL }
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-1 [... item-n]
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

PART I

Creating and Running EPFs

1

Overview of BIND and EPFs

This book introduces you to BIND and EPFs. BIND is Prime's utility for linking V-mode and I-mode programs. The runfiles that BIND creates from the object files produced by Prime's language translators are called EPFs (Executable Program Formats).

As you will discover, EPFs and BIND provide a new programming environment that was not available prior to Rev. 19.4.

Prime has two other utilities, SEG and LOAD, for creating programs from separate subroutines by linking them. The rest of this chapter describes the differences among BIND (the newest linker), SEG, and LOAD. If you are new to Prime systems, you may want to skip this chapter and go directly to Chapter 2. If you are familiar with SEG or LOAD, however, you may find this introduction useful.

PRIME'S LINKERS AND LOADERS

After you have written and compiled a program, you are ready to use a linker to create a runfile -- the executable version of your program. A linker resolves external references as it combines modules of a program. Thus, using one of the Prime linkers (BIND, SEG, or LOAD)

creates an executable file that consists of linked object files that contain one or more of the following:

- Subroutines
- User library routines
- System library routines
- Common areas

Linking Versus Loading

SEG and LOAD build a memory image of a program as they combine its modules. Therefore, these utilities are known as linking loaders. BIND, Prime's newest linking utility, performs only the linking function. PRIMOS itself performs the loading function for an EPF when the runfile is executed.

This difference provides substantial advantages for users of BIND. However, because the difference occurs inside the system, rather than being highly visible, BIND in many ways appears to be similar to LOAD and SEG. (BIND, however, is much simpler to use than SEG.) Because of this similarity, you may often see BIND called a loader. (Note that the subcommand for specifying a module as part of the runfile is still called LOAD in BIND, as it was in LOAD and SEG, even though the BIND subcommand only links the module, rather than both linking and loading the module.)

Addressing Modes

Another important distinction among Prime's three linkers concerns the types of modules they can combine. Prime systems support several different instruction sets. V mode and I mode are the most powerful, and differ primarily in performance for certain types of computations. Prime systems also support R mode, primarily for compatibility with programs written to run on Prime 100, 200, and 300 systems. Compared to V mode and I mode, R mode is a very limited instruction set. For example, R-mode programs can access only 128KB of memory, but V-mode and I-mode programs can access 32MB or more.

BIND and SEG link only V-mode and I-mode programs. The LOAD utility links only R-mode programs. Therefore, existing programs linked via the SEG utility can also be linked via BIND. However, existing programs linked via the LOAD utility must be converted from R mode to V mode (or from R mode to I mode) before they can be linked by BIND or SEG.

The LOAD Linking Loader

The first linking loader supplied by Prime was the LOAD utility. You use it to link and load R-mode programs (programs that run in 32R mode or 64R mode). Of the Prime-supplied language translators, only RPG, FTN, and PMA generate R-mode program modules. LOAD generates a runfile, called a static-mode runfile, that contains the linked program. Use the RESUME command to execute a program linked by LOAD.

For debugging an R-mode program, Prime provides the PSD utility (Prime Symbolic Debugger). This utility allows you to examine and change memory and set breakpoints at instructions in memory. You cannot use DBG, Prime's interactive source-level debugger, on R-mode programs.

Because R mode limits program size, speed, ease of debugging, and functionality, Prime provides methods for converting R-mode programs to V mode. RPG programs can be converted to VRPG; FTN programs can either convert to F77 (FORTRAN 77) or use a command line option to cause FTN to generate V-mode program modules; and PMA programs can be converted to using V mode or I mode by adding certain instructions and changing some existing instructions and data definitions.

If your program is written in R mode, and you do not intend to convert it to V mode or I mode, you must continue using the LOAD utility. See the SEG and LOAD Reference Guide for information on LOAD.

The SEG Linking Loader

The second linking loader supplied by Prime was the SEG utility. You use it to link V-mode and I-mode programs. Except for RPG, all of the Prime language translators generate V-mode program modules; some of them also generate I-mode program modules if you specify a command line option. SEG generates a runfile, called a SEG runfile, that contains the linked program. To execute a program linked by SEG, use the SEG command instead of the RESUME command.

SEG can also generate static-mode runfiles like the LOAD linker. This use, however, is considered an advanced use of SEG and it requires the use of special SEG commands and a special load sequence. The advantages of a static-mode runfile over a SEG runfile are that the former is invoked with the RESUME command and that it normally takes up less disk space.

For debugging a V-mode or I-mode program, Prime provides DBG, an interactive source-level debugger with many advanced features. DBG can operate on SEG runfiles, but not on static-mode runfiles generated by SEG or LOAD.

If your program is written in V mode or I mode (or in both), you may continue to use SEG, or you may convert your program to use BIND. If you choose not to convert it to BIND, you must continue using the SEG utility. See the SEG and LOAD Reference Guide for information on SEG.

The BIND Linker

The latest linker supplied by Prime is the BIND utility. You use it to link V-mode and I-mode programs. Except for RPG, all of the Prime language translators generate V-mode program modules; some of them also generate I-mode program modules if you specify a command line option. BIND generates a runfile called an EPF (Executable Program Format) that contains the linked program. To execute a program linked by BIND, use the RESUME command.

Prime's interactive debugger, DBG, operates on EPFs generated by the BIND linker.

Compared to SEG, BIND and the EPFs it creates make it easier for you to build and maintain software. The following is a list of benefits you gain when using BIND instead of SEG:

- The runfiles are directly RESUMEable.
- The RESUMEable runfiles can be debugged via DBG without having to be regenerated by using a different load sequence.
- You can use longer subroutine names. BIND limits entrypoint names to 32 characters, whereas SEG's limit is 8 characters and LOAD's limit is 6 characters.
- You can create your own libraries of entrypoints.
- PRIMOS takes care of memory allocation at program execution time. With SEG and LOAD, allocation is performed when the program is linked, thus limiting the placement of the program for execution.
- EPF runfiles normally use less paging disk space than their SEG file counterparts, because portions of the EPFs are paged in directly from their file system partition.
- PRIMOS can keep, or suspend, many EPFs in memory at one time for a user, without causing the EPFs to overwrite each other or destroy data.
- Program EPFs can be executed as commands or as command functions.
- Because the actual programming instructions in EPFs must be pure (that is, the program cannot modify its own instructions), PRIMOS automatically allows the runfiles to be shared among users. (Only the pure programming instructions are shared, not the data that the program uses.) To share programs with SEG requires that you and your System Administrator coordinate the use of shared segments. Such sharing is not needed with BIND; R-mode programs (linked with LOAD) cannot be shared at all.

- Because you have better access control over programs in memory, there is an improvement in security over sharing programs with SEG.

- It is easy to install a new version of a program even while it is still being used by other users; in this case, PRIMOS keeps a copy of the old version.

- All memory used by a program built with BIND is easily released either automatically by PRIMOS (some time after the program has terminated) or by the user via a PRIMOS command. After a program built with SEG is run, the memory it used is not released until the user logs out.

- With BIND, you can write new CPL command functions. These are EPFs that make use of an extended interface to the command processor, as described in the Advanced Programmer's Guide, Volume III: Command Environment.

- BIND keeps useful information in an EPF, such as the version of BIND used to link the program and the date and time the program was linked. In addition, BIND allows you to set a version number for your program and to place some information in a comment field, both of which are kept in the EPF and displayed by a PRIMOS command.

- You can obtain a map of a program built with BIND even if you just tried to run the program and encountered an error. The command that produces the map does not overwrite information on the error encountered by your program.

- A program built with BIND can call a new subroutine to execute PRIMOS commands, such as SPOOL, JOB, and LD.

Note

BIND is not another version of SEG. BIND has fewer commands than SEG. You can create an EPF with one PRIMOS command line using BIND; but, SEG does not allow you to do so. Anyone accustomed to working with SEG will find working with BIND easy. For example, a program loaded by using SEG's default options can be linked with BIND by giving almost the same command sequence that was given to SEG's VLOAD subprocessor. This process is described more fully in Chapter 2.

Most importantly, SEG creates static runfiles, with all the limitations that they imply, whereas BIND produces dynamic runfiles.

TYPES OF RUNFILES

Prime's linkers produce three types of runfiles:

- LOAD (and sometimes SEG) produce static-mode runfiles.
- SEG produces SEG runfiles, which are also static in nature.
- BIND produces EPFs, which are dynamic in nature.

Static and dynamic runfiles differ in the manner in which they are placed into memory. These differences are described below.

Static Runfiles

R-mode runfiles, created by LOAD, are static runfiles. They always execute in segment '4000 of PRIMOS. (The term segment refers to the way PRIMOS divides virtual memory.)

V-mode and I-mode runfiles created by SEG are also static. They normally execute in segment '4001 and in higher-numbered segments.

A program built as a static runfile has the following qualities:

- All segment locations used by the program are assigned at load time by you, by LOAD, or by SEG.
- The program uses the same segment(s) every time it executes.
- The program can be suspended and restarted by PRIMOS, but with the possibility that intervening static runfiles overwrite and destroy the data of the suspended program.
- The responsibility for managing memory allocation is left up to the loader (or to you, if it is shared code).
- The runfile for the program contains a complete image of the programming instructions (code) and the data that links modules (linkage).

Dynamic Runfiles

The EPF runfiles that BIND creates are dynamic. A program built as a dynamic runfile has the following qualities:

- The program can execute in any dynamic segment or segments. The program does not need to use the same segment each time it is invoked.

- The program is placed by PRIMOS into free (that is, unused) segments of your address space at runtime. Thus, PRIMOS manages the allocation of memory for you.
- Because the program can start at any point in memory and occupy any segments that are free, two or more programs can exist in memory concurrently. (That is, they can be suspended and restarted without the possibility of being overwritten, except by an errant program).
- The EPF runfile for the program contains an image of the programming instructions (code) and a description of the data that links modules (linkage) for a given program. PRIMOS uses this information when executing the EPF. Because PRIMOS knows where the linkage data is, PRIMOS can adjust the information to reflect the final placement of the program in memory.

EPFs that are run at different command levels normally do not conflict, because PRIMOS assigns an EPF only to segments that are not already in use by a suspended program. (Command levels, explained later in this chapter, represent a method by which PRIMOS allows you to enter system commands without resetting your program.)

For a more detailed look at the advantages of dynamic runfiles over static runfiles, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

TYPES OF EPFS

EPFs are of two types:

- Program EPFs
- Library EPFs

A program EPF contains a single main entrypoint and related subroutines that together constitute a program. A program EPF is invoked explicitly in one of three ways: by a user's invoking the RESUME command, by another EPF's calling the first EPF as a subroutine, or by a user's issuing a command that names a program EPF residing in the UFD CMDNC0.

A library EPF contains many subroutines, some (or all) of which are entrypoints to that library EPF. Unlike a program EPF, a library EPF is not invoked explicitly. Instead, a library EPF is invoked implicitly by another program when the program references an entrypoint within the library EPF.

Only program EPFs are discussed in this chapter. For further information on libraries and on library EPFs, see Chapter 6.

THE STAGES OF A PROGRAM EPF

The process by which a program EPF is created can be seen as a series of distinct stages. The following stages show the sequence involved in creating and executing an EPF:

Stage I — Create and enter your program into the system, using a text editor such as EMACS or ED. This text file is your source file. The name of the source file should have the appropriate suffix of the language in which it is written, such as .F77 for FORTRAN 77, .PL1G for PL1/G, .CBL for CBL, and so on.

Stage II — Compile your source file with a high-level language compiler or an assembler. For example, use the F77 compiler for FORTRAN 77, the CBL compiler for COBOL 74, and so on. For PMA modules, use the assembler, PMA. The compiler produces an object file (also called a binary file) from your source file. The object file contains information about your program in a form that BIND can understand and manipulate. The name of the object file is given the suffix .BIN if the source filename has a language suffix.

Stage III — Bind your program, using the BIND utility, to create the runfile. BIND does not actually place your program in memory at this point. Instead, BIND creates imaginary memory addresses that are turned into actual memory addresses by PRIMOS when you invoke the program. Binding the object files together generates a self-contained program EPF with the filename suffix .RUN in your directory. This is the linking stage.

Stage IV — Invoke the EPF by using the RESUME command. At this point, PRIMOS allocates free space in virtual memory for your program (based upon requirements contained in the program EPF file) and maps the program in the allocated space. This is the loading stage. PRIMOS then converts the imaginary addresses in the linkage data templates into actual addresses. Finally, PRIMOS begins program execution by calling the program as if it were a subroutine.

For a more detailed description of how an EPF is linked and executed, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

EPFS AND MEMORY ALLOCATION

As mentioned earlier, several EPFs can reside in memory at one time without overwriting or destroying each other's data. They can also coexist in memory with a single static-mode program. This capability allows EPFs to invoke other EPFs, static-mode programs, and PRIMOS commands as if they were subroutines. It also allows you to suspend one EPF, invoke another, and then resume execution of the first EPF. The way PRIMOS handles this situation is shown in Table 1-1.

Table 1-1
How PRIMOS Handles Suspended EPFs

WHEN YOU...	PRIMOS WILL...
Enter text using the editor, ED	Execute ED as an external static-mode program
Leave ED by using the BREAK key or typing a CONTROL-P	Suspend execution of your program
Type HELP for assistance	Execute HELP as a program EPF without overwriting previous EPFs or static-mode programs
Type a CONTROL-P again	Suspend execution of the HELP program
Type START	Reinvoke and continue execution of the HELP EPF
Finish using the HELP command	Return to the PRIMOS command level for the suspended ED program
Type START again	Reinvoke and continue execution of the ED program without overwriting or destroying data

There is a limit to the number of program EPFs you can have suspended at a time. This limit is discussed in Chapter 7.

SHARING PROGRAM EPFS

If two or more users run the same program EPF at the same time, PRIMOS:

- Automatically shares between the users the parts of the program that don't change
- Protects the shared parts of the program from being changed, by allowing the users to have only read and execute access

For more information on how PRIMOS takes care of sharing program EPFs, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

EPFS AND DBG

EPFs are handled properly by DBG, Prime's Source Level Debugger. If you are trying to find out why your program failed at execution time, follow these steps:

1. Compile your program with the -DEBUG option.
2. Link your program, using BIND.
3. Execute your program with the DBG command.

You can also execute the runfile thus created via the RESUME command. You do not need to use different BIND sequences to generate one runfile that can be invoked with DBG and one that can be invoked with RESUME. However, programs compiled with the -DEBUG option take up more disk space (due to the additional information supplied to DBG) and do not take advantage of any optimizations the compiler may perform. Therefore, you should eventually recompile the program without the -DEBUG option and relink it.

For more information, see the Source Level Debugger User's Guide.

REMOTE EPFS

If you invoke an EPF that resides on a remote system, it works almost exactly as if it were on the local system, with three differences:

- The EPF is not shared with other users on your system.
- The entire program is paged on the local system.
- The procedure code is not protected against writing by the user.

If the remote system shuts down after the program starts running, the program continues running without interruption.

2

Working With BIND

This chapter shows you how to use BIND to create an Executable Program Format (EPF). (Chapter 3 shows you how to use the RESUME command to execute EPFs.) For most of your linking needs, the basic subcommands given in this chapter are sufficient. These subcommands are:

Subcommand	Use
LOAD	Links one or more program modules.
LIBRARY	Links one or more libraries.
FILE	Stores the finished runfile on disk.
QUIT	Ends the BIND session without storing results.
MAP	Checks for unresolved references.
HELP	Displays information on BIND subcommands.
RELOAD	Relinks an object file into an existing EPF.

See Chapter 8 of this book for other BIND subcommands used for more specialized linking.

BEFORE YOU BEGIN...

Before you can use BIND, you must first compile your source program. This compilation process creates an object file (also called a binary file), which is the type of file you need as input to BIND.

Naming Files

Prime's convention for naming files allows you to readily identify file types and the relation of files to each other. A file named according to this convention has a two-part filename made up of a basename and a suffix. The suffix, which identifies the file's type, is separated from the basename by a period (.). For example, the filename CIRCLE.F77 has CIRCLE as its basename and F77 as its suffix.

The suffixes and the types of files that they identify are:

File Type	Suffix
Source file	The name of the compiler for the source file's language (examples: CBL, F77, PL1G, VRPG)
Object file	BIN (created by a compiler or the assembler)
Dynamic runfile	RUN (an EPF created by BIND)
SEG runfile	SEG (created by SEG)
Static-mode runfile	SAVE (created by LOAD or SEG)

You should follow this convention when you name your source files. Doing so makes your work easier because you can quickly tell in what language a source file is written. Proper suffixes also allow Prime's compilers to generate the .BIN suffix when they create your object files. This, in turn, allows BIND to operate most efficiently. For example, if you name a file TEST.F77 and then compile it, the F77 compiler automatically names the object file TEST.BIN. If you then use BIND to link TEST.BIN, BIND names the resulting runfile TEST.RUN.

Note

PRIMOS requires the .RUN suffix in order to to recognize and correctly handle an EPF. Therefore, BIND always adds the .RUN suffix to whatever default name it creates or to whatever name you choose for your runfile. If you later change the name of your runfile, make sure that you retain the .RUN suffix.

The following examples show the use of compiler-name suffixes for all of Prime's high-level languages:

Source filename	Compiler	Object filename
MYPROG.CC	CC	MYPROG.BIN
TEST.CBL	CBL	TEST.BIN
CIRCLE.FTN	FTN	CIRCLE.BIN
LIFE.F77	F77	LIFE.BIN
GAMES.PASCAL	PASCAL	GAMES.BIN
TARGET.PL1	PL1	TARGET.BIN
SAMPLE.VRPG	VRPG	SAMPLE.BIN

Filenames Without Suffixes: If your file does not have a compiler-name suffix, and you do not specify a particular name for your object file (via the -BINARY option), the compiler names your object file by adding the prefix B_ to the filename. For example, if you compile the file TEST, the compiler chooses a default filename of B_TEST for the object file. CC, CBL, FTN, F77, Pascal, PL/I and VRPG all use this naming convention. If you already have files named B_program, you may want to change their names to program.BIN.

Name Recognition: Both BIND and the compilers listed in this chapter recognize and search for appropriate suffixes. Recognizing appropriate suffixes is important for two reasons. First, it means that you do not have to type the suffix when invoking BIND. If you ask BIND to link the file TEST, BIND looks for a file named TEST.BIN (and then for a file named TEST.RUN) before it looks for plain TEST. The second reason is that BIND does not recognize prefixes, but instead considers them to be part of the base name of the file. If you have a file named B_TEST, you must specify its full name to BIND.

Table 2-1 shows how to create, link, and run a program whose object file uses the .BIN suffix. Table 2-2 shows how to link and run a program whose object file uses the B_ prefix.

Table 2-1
Creating and Running a Program With Suffixed Filenames

Step	Command	Action
Create a program	ED or EMACS	Give the program the filename basename.compiler-name. (Doing so indicates that the file contains a program of a specific language type.)
Compile the program	Compiler-name basename	(You do not need to specify the suffix.) This command generates an object module with the name basename.BIN.
Bind the program	BIND	The BIND command assumes the presence of the .BIN suffix, and creates a runfile with the name basename.RUN.
Run the program	RESUME basename	The RESUME command recognizes that the runfile is an EPF when it sees that it has the suffix .RUN.

Table 2-2
Linking and Running a Program Without Suffixed Filenames

Step	Command	Action
Create a program	ED or EMACS	Give your program the name filename (without a suffix).
Compile the program	Compiler-name filename	This command generates an object module with the name B_filename.
Bind the program	BIND	Tell BIND to link B_filename. Do not let BIND choose a default name for the runfile; specify a runfile name that does not have the B_ prefix. (As explained later in this chapter, you can give the runfile any name you choose; BIND still adds the .RUN suffix.)
Run the program	RESUME filename	PRIMOS recognizes the runfile as an EPF when it sees the .RUN suffix.

HOW TO USE BIND

After you have successfully compiled your program, you are ready to create a runfile -- the executable version of your program -- using BIND. There are two ways of using BIND:

- Directly, from the PRIMOS command line. (This is the command form.)
- Interactively, by issuing BIND subcommands. (This is the subsystem form.)

To use BIND properly, you should have at least the following access rights to your current directory: Add, Delete, List, Read, Use, and

Write. For example, if you use BIND in a directory in which you have only LUR rights, you receive the following message:

```
OK, BIND
[BIND rev 19.4]
Insufficient access rights.  T$0000 (TEMPF$)
OK,
```

Linking a Program With One Command

To invoke BIND from PRIMOS to link your program in one step, type the command:

```
BIND [EPF-filename] options
```

Invoking BIND in this way allows you to create your runfile with one PRIMOS command line. EPF-filename is the name of the EPF that you want BIND to create. BIND saves your runfile in your directory under the name EPF-filename.RUN. If you do not specify EPF-filename, BIND creates a filename by adding the suffix .RUN to the name of the first object file that you link, and saves the runfile in your directory.

The options that you give on the BIND command line correspond to internal BIND subcommands that are explained in the following sections. However, when used as command line options, the subcommands are prefixed with hyphens. Thus, LOAD becomes -LOAD, LIBRARY becomes -LIBRARY, and so on.

A sample BIND session using one PRIMOS command line looks like this:

```
OK, BIND MYPROG -LOAD ADD SUB -LIBRARY
[BIND rev 19.4]
BIND COMPLETE
OK,
```

In this example, BIND saves your runfile with the name MYPROG.RUN. The runfile contains the linked object files ADD.BIN and SUB.BIN, plus system library subroutines.

Running BIND Interactively

To invoke BIND interactively, type the command:

```
BIND [EPF-filename]
```

BIND then displays a colon (:) prompt. You respond to this prompt by issuing BIND subcommands. Each time you press the carriage return, you see this prompt. When you leave BIND, your system prompt appears on the screen.

A sample interactive BIND session using the same program as that used in the single-step example looks like this:

```
OK, BIND MYPROG
[BIND rev 19.4]
: LOAD ADD
: LOAD SUB
: LIBRARY
BIND COMPLETE
: FILE
OK,
```

BIND saves the runfile in your directory as MYPROG.RUN.

If you do not specify an EPF-filename when you use BIND interactively, BIND takes the name of the first object file that you link and adds the suffix .RUN. For example:

```
OK, BIND
[BIND rev 19.4]
: LOAD ADD
: LOAD SUB
: LIBRARY
BIND COMPLETE
: FILE
OK,
```

The runfile in your directory is given the name ADD.RUN.

BASIC SUBCOMMANDS FOR LINKING WITH BIND

Essential Subcommands

You can link most of your programs with the subcommands LOAD, LIBRARY, and FILE. If you are using BIND interactively, after you receive the colon (:) prompt, enter the subcommands in the following sequence:

1. Use the LOAD subcommand to link your program, starting with the main procedure and followed by subprograms in any order.

2. Use the LIBRARY subcommand to link the libraries you need. (See Table 2-3.)

3. Use the FILE subcommand to save the EPF runfile and return to PRIMOS.

Other Useful Subcommands

At times, you may want to use the following subcommands. You may use them at any time during the linking sequence.

- Use the QUIT subcommand to return to PRIMOS command level without saving the current EPF. (This subcommand aborts the BIND session.)

- Use the MAP -UNDEFINED subcommand to identify unresolved references (subroutines that are called but that you have not yet linked) if you do not receive a BIND COMPLETE message at your terminal when you expected it.

- Use the HELP subcommand to get online help on the subcommands if you run into trouble while using BIND.

If you are using BIND on the command line, each of the above subcommands must be preceded by a hyphen (-).

After Your EPF Is Created

It sometimes happens that, when you have created and run your EPF, you find problems in one or more modules. In these cases, BIND allows you to use a quick linking sequence, using the LOAD and RELOAD subcommands, to link a corrected version of one or more modules into the existing EPF. The procedure for doing this is shown at the end of this chapter.

Using the LOAD Subcommand

The LOAD subcommand links object files into the EPF. It has the following format:

```
LOAD pathname-1 [pathname-2 pathname-3...]
```

This subcommand links the files specified by pathname to the current EPF. If you have not yet named the EPF by specifying its name on the BIND command line, BIND uses the name you give in the first pathname. Each pathname is the name of an object file (binary file). When two or more subroutines are present in an object file, the subroutines are linked in the order in which they appear in the object file.

Examples:

```
: LOAD HOME                        /* Links object file
                                      with the name HOME.BIN.
                                      The runfile in your
                                      directory is HOME.RUN.

: LOAD HOME ADD SUB                /* Links object files
                                      HOME.BIN, ADD.BIN,
                                      and SUB.BIN. The
                                      runfile in your
                                      directory is called
                                      HOME.RUN.
```

Using the LIBRARY Subcommand

The LIBRARY subcommand links system libraries into the EPF. It has the following format:

```
LIBRARY [library-name-1 library-name-2 library-name-3...]
```

This subcommand links application libraries, language libraries, and other system libraries. The subcommand format is the same as that of the LOAD subcommand. The only difference is that all libraries (except those with pathnames) are obtained from a directory called LIB, in which all of Prime's standard system libraries are kept. If you are linking a library supplied by you or by someone else, the LIBRARY subcommand looks for the file in the location you specify in the pathname.

During each BIND session to link a program, you must issue the LIBRARY subcommand with no pathname specified, in order to link the system library. This procedure links the PFTNLB.BIN library in the LIB UFD.

Typically, it is the last command you use to link a module (via LOAD or LIBRARY) during a BIND session.

Examples:

```
: LIBRARY TEST>MYLIB        /* Gets the file MYLIB.BIN from
                               the TEST UFD.

: LIBRARY VFORMS            /* Gets the library VFORMS.BIN
                               from the LIB UFD.

: LIBRARY                   /* Gets the file PFTNLB.BIN from
                               the LIB UFD.
```

You must link the libraries in the following specific order:

1. The libraries supplied by you.
2. The libraries that are specific to each language.
3. The application libraries.
4. The system libraries.

User-supplied Libraries: These are libraries supplied by you, the user, in a file that you specify in the pathname when you use the LIBRARY subcommand. They are not usually kept in the directory LIB. (You may also link these libraries with the LOAD subcommand.)

Language-specific Libraries: All Prime languages, except for FORTRAN and PMA, require that a specific language library be linked. See Table 2-3 for the library you need to use for each language. You may link more than one language library if your program is made up of subroutines written in more than one language.

Examples:

```
: LIBRARY PASLIB            /* Links the PASCAL language library

: LIBRARY CBLLIB            /* Links the CBL language library
```

Table 2-3
Language Libraries

Language	Library
C	C_LIB
CBL(COBOL 74)	CBLLIB
F77	-
FTN	-
Pascal	PASLIB
PL1G	PL1GLB
PMA	-
VRPG	VRPGLB

Application Libraries: If you are using subroutines from Prime's applications library or are sorting and merging subroutines from one of the sort libraries, you must use the LIBRARY subcommand followed by VAPPLB or VSRTLI to link these special libraries. If your CBL program uses MIDASPLUS files, you must link the VKDALB library with the LIBRARY subcommand. You may link as many application libraries as your program uses.

Examples:

```
: LIBRARY VAPPLB            /* Links the applications library

: LIBRARY VSRTLI            /* Links one of the sort libraries
```

Standard System Library: All programs require the standard system library. Use the LIBRARY subcommand with no arguments, in order to link the standard system library.

Example:

```
: LIBRARY                   /* Links the standard system library
```

After the standard system library has been linked, BIND COMPLETE appears on your terminal. If you do not receive this message, use the MAP -UNDEFINED subcommand to identify unresolved references.

Using the FILE Subcommand

After you receive the message BIND COMPLETE at your terminal, use the FILE subcommand. It has the following format:

```
FILE [EPF-filename]
```

After you issue this subcommand, BIND:

1. Performs some final processing of the EPF.
2. Files the EPF in your directory with a .RUN suffix. If you specified a name for the EPF, it is written in your directory under the name EPF-filename.RUN.
3. Returns control to PRIMOS.

Example:

```
OK, BIND
[BIND rev 19.4]
: LOAD SHORTY
: LIBRARY
BIND COMPLETE
: FILE TEST
OK,
```

BIND files the runfile in your directory under the EPF-filename of TEST.RUN. If you already have an EPF with the same name, BIND usually overwrites the existing EPF. However, if other users are already using the existing EPF or if you suspended the existing EPF while running it and did not release it, BIND does not overwrite the existing EPF. Instead, BIND changes the suffix of the name of the existing EPF from .RUN to .RPn, where n is a digit (0 through 9). Then, BIND writes out your EPF with the suffix .RUN. That way, users already using the existing version of your EPF are not affected by the installation of the new version.

For example, if the SHORTY program, linked in the previous example, is in use when you attempt to generate a new version of it, BIND changes the name of the old version from SHORTY.RUN to SHORTY.RP0 before creating the new version (SHORTY.RUN).

In this case, the BIND session looks like this:

```
OK, BIND
[BIND rev 19.4]
: LOAD SHORTY
: LIBRARY
BIND COMPLETE
: FILE
The file is in use.  The old file is now called SHORTY.RPO.
OK,
```

Note

A filename specified by the FILE subcommand overrides any filename specified on the BIND command line.

Default FILE: If you are using the single-step BIND invocation from the PRIMOS command line, you do not have to specify the -FILE option. In command form, BIND assumes you wish to FILE the EPF as the last step, unless you specify the -QUIT option.

For example:

```
OK, BIND -LOAD HOME ADD SUB -LIBRARY
[BIND rev 19.4]
BIND COMPLETE
OK,
```

In this example, BIND automatically appends a -FILE option and saves the runfile in your directory under the EPF-filename HOME.RUN.

Therefore, if you are using the command form of BIND and you do not want the EPF to be created, you must use the -QUIT option to end the command line, as shown in the next section.

Using the QUIT Subcommand

If you want to leave BIND without completing the link process or if you do not want an EPF to be made while using the single-step command line linking, use the QUIT subcommand:

```
QUIT
```

The QUIT subcommand ends a BIND session without saving the current EPF. BIND asks you to verify that you have not filed the EPF before

returning to PRIMOS. If you already have an EPF with the same name as the one you are building during the current session, using the QUIT subcommand to abort the BIND session does not overwrite the existing runfile.

The following example illustrates the QUIT subcommand during an interactive session:

```
OK, BIND
[BIND rev 19.4]
: LOAD PAY
: LIBRARY
BIND COMPLETE
: QUIT
EPF not filed, ok to quit? ('Yes','Y','No', 'N'): YES
OK,
```

In this example, if an EPF named PAY.RUN already exists, it is not overwritten, because the runfile was not filed. If you answer NO to the verification query, you remain in BIND and the colon prompt is displayed again.

Using QUIT as a command line option is shown in the following example, which uses the same PAY.BIN file:

```
OK, BIND -LOAD PAY -LIBRARY -QUIT
[BIND rev 19.4]
BIND COMPLETE
OK,
```

The result of this example is the same as the previous one.

Note

Users familiar with SEG should note the difference between the QUIT subcommands of BIND and SEG. The QUIT subcommand of SEG writes out the SEG runfile (overwriting an existing runfile in the process) before returning to PRIMOS. In BIND, the QUIT subcommand does not write out the EPF runfile before returning to PRIMOS. Whereas SEG overwrites an existing SEG runfile as soon as you begin loading modules into it, BIND does not overwrite an existing EPF runfile until you issue the FILE subcommand.

Using the MAP -UNDEFINED Subcommand

If you did not receive the message BIND COMPLETE at the end of your BIND session, you can use the MAP -UNDEFINED subcommand to find out whether you have any unresolved subroutine references. To do this, issue the subcommand:

```
MAP -UNDEFINED
```

For example:

```
OK, BIND
[BIND rev 19.4]
: LOAD WHO
: LIBRARY
: MAP -UNDEFINED

Map of WHO

UNDEFINED SYMBOLS:
  XIT       -0002/000020
: QUIT
EPF not filed, ok to quit? ('Yes','Y','No', 'N'): YES
OK,
```

Here, the user realizes that a reference to a subroutine named XIT in the program WHO should have been a reference to EXIT instead. The user issues the QUIT subcommand. In such a situation, you would normally edit your program, correct the misspelling, and recompile your program before attempting to link it again.

Other possible causes of unresolved references are that you did not link one of the subroutines your program uses or that you did not link one of the libraries your program uses.

Table 2-4 lists some examples of unresolved references and what you can do about them.

Table 2-4
Unresolved References

Problem	Solution
Missing routines that should have been linked.	Return to the point in the BIND session where the missing routines should have been linked; link them; then proceed through the BIND session again until the message BIND COMPLETE appears.
Misspelling of a reference in the source code.	Correct the error in the source code; recompile and relink the program.
Program error caused an identifier to be misinterpreted or not generated by the compiler.	Correct the error in your program; recompile and relink the program.

Later in this chapter, there is a section that explains what happens during an ordinary program linking sequence. There you can find examples illustrating what to do if you do not receive a BIND COMPLETE message.

Using the HELP Subcommand

If you run into trouble during your BIND session, use the HELP subcommand in response to the colon (:) prompt. The HELP subcommand has the following format:

```
HELP  { subcommand-name }
      { -LIST           }
```

The HELP subcommand displays some brief information on the syntax and semantics of a specific subcommand if you type HELP followed by the subcommand name. Type HELP -LIST for a list of all BIND subcommands.

For example:

```
: HELP FILE
FILE [<epfname>]
     will exit to PRIMOS after filing the EPF.
     If <epfname> is specified, the EPF will be named <epfname>.RUN
```

EXAMPLES OF A STANDARD LINKING SEQUENCE USING BIND

The following example shows a standard linking sequence using BIND. In response to the colon (:) prompt, you link a CBL program.

```
OK, BIND
[BIND rev 19.4]
: LOAD SAMPLE3                   /* Link program first
: LIBRARY CBLLIB                 /* Link the language-specific
                                    library
: LIBRARY                        /* Link the standard libraries
BIND COMPLETE                    /* No more unresolved references
: FILE                           /* Save the EPF and return to
OK,                              /* PRIMOS
```

You can accomplish this same sequence in a single step by giving the required subcommands in the PRIMOS command line that invokes BIND. When you use this method, you must precede each BIND subcommand with a hyphen (-).

```
OK, BIND -LOAD SAMPLE3 -LIBRARY CBLLIB -LIBRARY
[BIND rev 19.4]
BIND COMPLETE
OK,
```

RETRYING A LINKING SEQUENCE

Suppose that, during your binding session with a Pascal program, you forget to link the standard system library. When you do not receive a BIND COMPLETE message, you check for the missing reference and relink from the point at which the omission occurred. Your binding sequence looks like this:

```
OK, BIND
[BIND rev 19.4]
: LOAD PASS                     /* Link the main program.
: LOAD VAL                      /* Link the called program.
: LIBRARY PASLIB                /* Link the Pascal library.
: LIBRARY PL1GLB                /* Link the PL1G library.
: MAP -UNDEFINED                /* Use the MAP command to
Map of PASS                        find unresolved references.

UNDEFINED SYMBOLS:
  P$ASBC   -0002/000072
  P$ENTP   -0002/000070
: LIBRARY                       /* Link the missing standard
BIND COMPLETE                      system libraries.
: FILE                          /* File the EPF and return
OK,                                to PRIMOS level.
```

In the following F77 program, there is an error in the spelling of a reference in the source code, and the error does not show up during the compilation process. However, you do not receive a BIND COMPLETE message when you try to create an EPF. The MAP -UNDEFINED subcommand indicates that there is an undefined symbol named XIT. You use the QUIT subcommand to leave BIND (without saving the EPF); correct the error in the source code; recompile the program; and relink it, using BIND.

```
OK, BIND
[BIND rev 19.4]
: LOAD HOME                     /* Link main program.
: LIBRARY                       /* Link the libraries.
: MAP -UNDEFINED                /* Use the Map command to
Map of HOME                        help locate the error.

UNDEFINED SYMBOLS:
  XIT    -0002/000030
: QUIT                          /* Use the QUIT command
                                   to leave BIND.

EPF not filed, ok to quit? ('Yes','Y','No', 'N'): YES
OK,
```

After you correct the error in your program, your compiling and linking sequence looks like this:

```
OK, F77 HOME                       /* Recompile your program.
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77-REV 19.4]

OK, BIND
[BIND rev 19.4]
: LOAD HOME                        /* Link the main program.
: LIBRARY                          /* Link the libraries.
BIND COMPLETE                      /* Successful link!
: FILE                             /* File the EPF and
OK,                                /* Return to PRIMOS.
```

RELOADING A MODULE

Suppose you have created an EPF named BIG_PROGRAM, which contains six program modules. Suppose also that when you run this program you discover an error in one of the modules, MODULE_5. You correct the error and recompile MODULE_5. You now want to rebind the program and re-run it to see that it is now working correctly.

To do this most quickly, you first link the existing EPF, then relink your new version of MODULE_5. The BIND sequence looks like this:

```
OK, BIND
[BIND rev 19.4]
: LOAD BIG_PROGRAM.RUN
: RELOAD MODULE_5
BIND COMPLETE
: FILE
OK,
```

If you do not get the BIND COMPLETE message after relinking your module, you should reissue the LIBRARY commands that you used in your original BIND sequence. This situation would happen if you added new library calls when you rewrote the module.

For more details on the RELOAD command, see Chapter 8.

3

Running EPFs

This chapter describes how to run an EPF (Executable Program Format) by using the PRIMOS level RESUME command. For more language-specific information, see the appropriate language reference guide.

RUNNING YOUR PROGRAM

Once you have compiled and created an EPF, you are ready to run your program, using the RESUME command. The RESUME command has the following format:

```
RESUME pathname [program-arguments]
```

program-arguments are arguments that are passed to the program.

For example, suppose you have used BIND to create a runfile with the name MYPROG.RUN. To execute MYPROG, use the PRIMOS level command:

```
RESUME MYPROG
```

PRIMOS automatically looks in your directory for a file called MYPROG.RUN. If it finds such a file, PRIMOS begins executing it as an

EPF runfile. If it does not find MYPROG.RUN, PRIMOS then searches for the following files, in order, and executes the first that it finds.

1. MYPROG.SAVE (static-mode runfile generated by LOAD or SEG)

2. MYPROG.CPL (CPL program)

3. MYPROG (static-mode runfile generated by LOAD or SEG)

If PRIMOS finds none of these files, it displays the message:

```
Not found. MYPROG (std$cp)
ER!
```

For more information on running programs see the Prime User's Guide.

SAMPLE SESSIONS

The following sample session uses an F77 program called POWERS to show you how to create and execute an EPF. The program displays a table of numbers, one through five, and calculates the square, cube, and fourth power for each number. First, you compile the program:

```
OK, F77 POWERS
[F77 Rev. 19.4]
0000 ERRORS [<.MAIN.> F77-REV 19.4]
OK,
```

Then, using BIND from the command line, you create an EPF:

```
OK, BIND -LOAD POWERS -LIBRARY
[BIND rev 19.4]
BIND COMPLETE
OK,
```

You now have in your directory a file with the filename POWERS.RUN. This is what happens when you execute your program:

```
OK, RESUME POWERS

          NUMBER        SQUARE        CUBE        FOURTH

             1             1            1            1
             2             4            8           16
             3             9           27           81
             4            16           64          256
             5            25          125          625

OK,
```

In this next session, you have a COBOL 74 program that adds and subtracts from an initial value. Once again, the first thing you must do is to compile the program:

```
OK, CBL VALUES

[CBL rev 19.4]
OK,
```

After you compile your program successfully, you invoke BIND to create an EPF:

```
OK, BIND
[BIND rev 19.4]
: LOAD VALUES
: LIBRARY CBLLIB
: LIBRARY
BIND COMPLETE
: FILE
OK,
```

In your directory there is a runfile called VALUES.RUN. To begin execution of the EPF, type:

```
OK, RESUME VALUES

THIS IS A PROGRAM TO ADD AND SUBTRACT FROM AN INITIAL TOTAL

WHAT IS INITIAL VALUE OF TOTAL?
      **  NOTE FORMAT MUST NOT USE DECIMAL POINT.
      **  EX: TO REGISTER $45.25, ENTER 4525.
3695
ENTER AMOUNT, PRECEDED BY :  A FOR ADDITION
                             S FOR SUBTRACTION
                             Q FOR QUIT
S2645
BALANCE SO FAR:    10.50
ENTER CODE AND AMOUNT (Q TO QUIT).
Q
BALANCE IS:
    10.50
OK,
```

PART II

Using BIND and EPFs

4

Programming With EPFs

OVERVIEW

In the first three chapters of this book you learned about EPFs and the fundamentals of using BIND and RESUME to create and execute them. This section of the book is about programming with BIND and EPFs and is divided into the following chapters:

- Chapter 4 describes the limitations and restrictions necessary when you program with EPFs.
- Chapter 5 describes what happens when you convert an existing static-mode program to an EPF.
- Chapter 6 describes libraries and how to create EPF libraries.
- Chapter 7 describes what to do if you run into problems when you use EPFs.

This chapter describes the programming environment necessary for building EPFs. The following topics are covered:

- Programming with EPFs versus static-mode programs
- Guidelines for programming with EPFs
- Calling other programs

Note

This chapter describes only program EPFs. For information on building and using library EPFs, see Chapter 6 and the Advanced Programmer's Guide, Volume I: BIND and EPFs.

PROGRAMMING WITH EPFS VERSUS STATIC-MODE PROGRAMS

Programming with EPFs is much easier than programming with static-mode programs because you do not have to worry about memory allocation. PRIMOS automatically takes care of this for you at execution time.

EPFs give you a choice of writing large applications in one of the following two methods:

- The application may be one program image.
- The application may be separated into EPF programs and EPF libraries. (EPF libraries are discussed in Chapter 6.)

You will find that building program applications as EPFs is easier and faster than building them as static-mode programs.

You will also find it easy to convert existing static-mode programs to EPFs. Indeed, many static-mode programs can be converted to EPFs simply by relinking their object files with BIND. To decide whether a given program is suitable for conversion, read the guidelines for programming EPFs, given in the remaining sections of this chapter, and the information about converting static-mode programs to EPFs, given in Chapter 5.

GUIDELINES FOR PROGRAMMING WITH EPFS

When programming with EPFs, you should follow these guidelines in order to meet the requirements of the EPF format:

1. Use only pure code in your programs. Prime's higher-level language compilers produce only pure, reentrant code. (Reentrant code means that the code does not change during program execution; it is not self-modifying.)

2. If you are programming in Prime's Macro Assembler language, PMA, make sure to separate your non-constant data areas (ECBs, linkage data, and common data) from your executable code by using the LINK and PROC pseudo-ops.

   It is best to write pure code in PMA, as in other languages. However, EPFs written in PMA can use impure code. If you do use impure code in a program, you must make sure to flag it via the IMPURE option of the SEG or SEGR pseudo-op.

3. Avoid using the RDTK$$ and COMANL subroutines to read information from the command line. Code the main routine of your program to accept the command line as an argument rather than having RDTK$$ retrieve command-line information. Also, do not use COMANL; instead, have your program call the CL$GET subroutine to input a new command line from the terminal. Your program may then parse the command line via the CMDL$A or CL$PIX subroutine. For more information about these routines, see the Subroutines Reference series.

   Any EPF that expects arguments when invoked should have the command line as the first of its main entrypoint arguments. For information on accepting command lines as arguments to EPFs, see Chapter 10.

4. Avoid using the PRIMOS subroutines that you would use to terminate your static-mode programs:

   - EXIT
   - ERRPR$ with the K$NRTN key
   - ERRSET

   If you use these subroutines, your program cannot be restarted. Instead, use an appropriate return or stop statement. Appropriate statements for Prime-supplied language translators are:

Language	Return Statement
C	return or exit()
COBOL, CBL	STOP
F77, FTN	RETURN or STOP
PL/I	RETURN, STOP, or END
Pascal	END
PMA	PRTN

5. To pause and then restart a program, use one of the following:

   - A PAUSE statement
   - The SIGNL$ subroutine with the PAUSE$ condition
   - The PL/I SIGNAL statement

- The ERRPR$ subroutine with the K$SRTN key
- The COMLV$ subroutine

  Any of the above pauses your program, creates a new command level, and allows the suspended program to be restarted via the START command in the same way that static-mode programs are restarted.

6. Do not use the -PBECB option with the FTN compiler. If you BIND an FTN program that was compiled with this option, BIND produces the following warning message:

   ```
   Warning: ECB MYPROG loaded into PROC segment.
   ```

   If you receive this message, the program will not execute.

   When used with compilers other than the FTN compiler, the -PBECB option causes the procedure code for a module to be treated as impure by BIND. Impure modules cannot be shared among users, but a program containing impure modules will execute.

7. Avoid using the GETERR, PRERR, and ERRSET subroutines to set or retrieve error information. Instead, you should do one of the following: use PRIMOS subroutines that return error codes (instead of older versions that set the static error vector); use the ERRPR$ subroutine to display error messages; or pass returned error codes as an argument. For more information about this last method of handling error information, see Chapter 10.

For additional information about these guidelines, about writing EPFs in PMA, and about EPF programming techniques in general, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

CALLING PROGRAMS

EPFs, CPL programs, and static-mode programs can call other programs while executing without returning to command level and without affecting the calling program's environment. When the called program finishes, execution continues within the initial (calling) program.

What Programs Can EPFs Call?

An EPF can call another EPF that can call yet another EPF. There is only one limit on the number of programs that can be called in this

fashion (program A calling program B, which calls program C, and so on). This imposed limit is called the command environment breadth limit and is set by your System Administrator. (To find out what your limit is, use the LIST_LIMITS command described in Chapter 9.) There are no other limitations on EPF programs' calling other EPF programs.

EPFs can call CPL programs to the same level of breadth that they can call other EPFs.

EPFs can also call static-mode programs. The following limits apply:

- Only one static-mode program may be active at any time.
- A program EPF can call a static-mode program, which can then call other program EPFs. However, the latter program EPFs cannot call static-mode programs.

What Programs Can CPL Programs Call?

CPL programs can call and be called just like EPFs. CPL programs and program EPFs share many characteristics; they can call each other, subject only to the restrictions on their use of static-mode programs and on the command environment breadth limit set by the System Administrator.

What Programs Can Static-mode Programs Call?

Static-mode programs can also call EPFs. The following rules apply:

- Static-mode programs can call EPFs.
- Static-mode programs can call CPL programs.
- Static-mode programs cannot call EPFs or CPL programs that call other static-mode programs.
- Static-mode programs cannot call static-mode programs because the second program overwrites the first.

For information about the routines that allow programs to call other programs, see Chapter 10.

Limitations Involved With Static-mode Programs

The major limitation, which lies at the root of all other limitations, is that you can have only one static-mode program in memory at any given time. Each succeeding static-mode program overwrites its predecessor.

Take, as an example, a CPL program named DISPLAY.CPL that invokes SLIST, a static-mode command. If a static-mode program, CALL_DISPLAY, calls DISPLAY.CPL, all is well up to the point where DISPLAY.CPL invokes SLIST. At that time, SLIST overwrites CALL_DISPLAY. When DISPLAY.CPL finishes running, it cannot return to its caller (CALL_DISPLAY) because its caller no longer exists.

Because of this behavior, we can see that DISPLAY.CPL, even though it is not in fact a static-mode program, behaves like a static-mode program because it causes the overwriting of a previous static-mode program.

A general rule of thumb, therefore, is that a program behaves like a static-mode program if:

- It is in fact a static-mode program.
- It is an EPF that calls any program that behaves like a static-mode program.
- It is a CPL program that calls any program that behaves like a static-mode program.

(Note that this is a recursive definition.)

Any program that behaves like a static-mode program cannot be called by another static-mode program.

5

Converting to EPFs

This chapter shows you how to convert your existing static V-mode and I-mode programs to EPFs. The following topics are covered in this chapter:

- Reasons for converting your programs
- Programs that are not convertible
- Conversion procedures for programs
- Conversion procedures for complex SEG load sequences

WHY DO I WANT TO CONVERT TO EPFS?

There are several reasons for converting your existing V-mode and I-mode programs to EPFs. As EPFs, they:

- Can be RESUMEd.
- Can be used as commands.
- Can be as large as you want -- you do not have to worry about space limitations. (There are limits defined by your System Administrator.)
- Will not overwrite each other in memory.

- Can be called from other programs.
- Are very easily built, using BIND.
- Are automatically shared among users. In addition, users who do not have access to a runfile containing a program EPF also cannot access the shared copy of the program.
- Allow you to use subroutine names longer than 8 characters.
- Can be both debugged via DBG and RESUMEd without requiring two versions built using two different link sequences.
- Can contain program version information and a comment field, such as for a copyright notice.
- May start up faster after a RESUME command, particularly if they were kept as SEG runfiles previously. This ability, however, depends on several factors, such as how much linkage data the program contains.
- Can disable or alter the behavior of command preprocessing features such as wildcarding, treewalking, iteration, and name generation.
- Can directly invoke PRIMOS commands, such as SPOOL, JOB, SLIST, and ED.

The reasons for converting your static V-mode and I-mode programs to EPFs are examined in the following sections.

EPFs and PRIMOS

EPFs are a special type of file format. An EPF is a Direct Access Method (DAM) file that contains both an image of the procedure and a description of the linkage for a given program. Previously, the static-mode SEG runfiles and the static-mode .SAVE runfiles represented only a memory image of both procedure and data. Following are the advantages of having the EPF contain the procedure and the description of the data for your program:

- PRIMOS automatically builds the linkage and resolves all addressing requirements for you at runtime.
- Procedure code (read-only programming instructions) is paged in directly from the file system partition instead of from a copy on the paging partition as is done for static runfiles. Direct paging gives better performance.

Suspending EPFs

Because PRIMOS places your EPF into areas of memory that it finds free at runtime, you can keep or suspend several EPFs in memory at the same time. You do not have to worry about these programs overwriting each other and destroying each other's data, unless one or more of them are errant programs.

EPFs Calling EPFs

With EPFs, an executing program can call another program without returning to command level and without affecting the calling program's environment. When the called program finishes, execution continues within the initial program. You can use this facility to:

- Execute an external PRIMOS command from within a program
- Start another subsystem from within the current program
- Use collections of simpler programs to do more complex jobs

Building EPFs With BIND

Using BIND to create dynamic runfiles (EPFs) is easy because much of the work is taken care of automatically by PRIMOS. In addition, BIND can be used either interactively or directly from the PRIMOS command line.

EPFs and the Command Processor Stack

When you build an EPF, you issue no instructions about where the stack for the EPF is to be placed, as is sometimes necessary when you use SEG. For the EPF stack, PRIMOS uses the same stack used by the command processor. To PRIMOS, the EPF looks and acts just like any other subroutine.

CONVERSION RULES

The conversion of programs written in Prime's high-level languages usually consists of linking the object modules with BIND instead of with SEG. For PMA programs, however, special attention must be given to the contents of each program. For information about how to write PMA programs for use in an EPF, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

WHAT CAN'T I CONVERT?

There are some programs that are not convertible to EPFs. You cannot convert a static-mode program to an EPF if the program:

- Is an FTN program compiled with the -PBECB option
- Has self-modifying (impure) code and is not a PMA program beginning with the SEG IMPURE or SEGR IMPURE pseudo-op
- Uses CALL EXIT as a Pause function
- Shares linkage segments
- Fails to declare all of the memory (stack, linkage, and procedure) that it uses
- Is in V or I mode and uses R-mode or S-mode subroutines
- Expects the initial values of the A, B, or X registers to be defined
- Depends on automatic initialization

FTN Programs Using the -PBECB Option

When you compile an FTN program with the -PBECB option, the resulting object files share ECBs with executable code, mixing writeable storage with read-only storage. (With EPFs, ECBs are modified when the program is run because they contain pointers to link frames. The locations of these link frames are not known until the program is run.) Because EPFs have read-only procedure code, they cannot make full use of object files that have been compiled with this option. In most cases, recompiling the program without the -PBECB option allows the program to be bound as an EPF.

Routines With Modifiable Data in the Program

Programs containing self-modifying code cannot be converted to EPFs, unless they are PMA programs that begin with the SEG IMPURE pseudo-op (for V-mode programs) or with the SEGR IMPURE pseudo-op (for I-mode programs). Self-modifying code can change while a program is running. EPFs must be pure code (that is, code that does not change while it is running).

This restriction is most likely to apply to PMA programs, many of which use impure code. However, it holds true for programs in any language. An impure PMA module that uses SEG IMPURE or SEGR IMPURE can be linked into an EPF, but its procedure code is not shared among users and it is not paged directly from the file system disk. Such a module is treated

similar to static data, incurring higher overhead costs for executing the EPF.

Note that it is frequently possible to rewrite impure programs so that they contain only pure code. After being rewritten in this fashion, the programs can be converted.

Use of CALL EXIT as a Pause Function

The EXIT subroutine in PRIMOS is most often used by a program to signal its completion. However, because static-mode programs remain in memory, PRIMOS allows a START command, issued after a static-mode program terminates via a call to EXIT, to continue program execution immediately after the call to EXIT.

If your program depends upon the ability to be restarted after a call to EXIT, you must first change the call to EXIT to be a call to COMLV$ instead. Otherwise, when your program calls EXIT as an EPF, PRIMOS treats the call as a permanent termination, and a subsequent START command does not continue the execution of the program.

Note that repeated execution of a program that calls COMLV$ without issuing the START or RELEASE_LEVEL commands between each execution may result in messages such as:

```
Now at command level 5. To release use RLS. (listen_)
```

If the user ignores these messages and continues running a program that uses COMLV$ to exit, he or she will ultimately reach the limit of command levels set by the System Administrator, and will be placed in mini-command level. Chapter 7 describes mini-command level.

Programs Sharing Linkage Segments

Libraries in which some linkage has been made public cannot be converted to EPFs. (Public means that the linkage resides in a shared segment.) Making linkage public is usually done to minimize the working set of a library. However, with EPFs, it is not possible to put linkage into shared segments.

Failure to Declare Memory Used

Some static-mode programs allocate less common area, link-frame space, or stack-frame space than they actually use. For example, some programs declare arrays in common areas to be one element long, and then proceed to use many more elements in the array. Some programs also declare arrays in stack or linkage frames to be shorter than they actually are. Built as static-mode programs, such programs may execute successfully despite these bugs; built as EPFs, these bugs may suddenly prevent the program from operating correctly.

Misdeclared Common Area or Link Frame: For example, a misdeclared common area or link frame might not result in any adverse program behavior when the program runs as a static-mode program, because the memory used by the misdeclared area might be loaded as the last area of a segment. Use of memory beyond the declared portion of the area simply causes more of the segment to be used; only when the segment overflows does the program misbehave.

However, built as an EPF, the program might share the segment that contains its link frame or common area with one or more other programs. In that case, when the program writes beyond the end of the declared area, it may overwrite portions of another program. This is a subtle error, because the errant program may in fact appear to work; the overwritten program, when next run by the user, may suddenly fail to work. (A clue to this occurrence is the fact that the overwritten program worked the first time it was run, at which point it was first loaded into memory; running it the second time causes PRIMOS to reuse its memory image if it is still in memory, and by this time, portions of it may have been overwritten.)

Misdeclared Stack Frame: Another example is a subroutine that misdeclares the size of its stack frame. If it calls only internal PRIMOS subroutines, or if calls to other subroutines occur just prior to the return of the errant subroutine, it may work under most circumstances as a static-mode program. (Again, only if its misdeclared stack size causes its true stack frame to overflow a segment will the program misbehave.) The reason the subroutine may work is because the invocation of the dynamic linking mechanism takes place on a different stack (the command processor stack) and hence does not overwrite part of the errant subroutine's stack frame. In addition, many of the internal PRIMOS subroutines (such as TNOU, TNOUA, SRCH$$, and so on) also run on a different stack.

However, when a subroutine that misdeclares its stack frame size is linked into an EPF, it may suddenly fail just after calling its first subroutine if that subroutine is external to the EPF (a dynamic link). This failure occurs because the dynamic linking mechanism runs on the same stack as the EPF — the command processor stack — and hence its data will probably overwrite portions of the data in the errant subroutine's stack frame. The behavior of the errant subroutine at this point is entirely unpredictable, but typically the result is the

signaling of a POINTER_FAULT$, ACCESS_VIOLATION$, ILLEGAL_SEGNO$, or OUT_OF_BOUNDS$ condition.

If Your Program Misdeclares Memory: Therefore, if you suspect that your program or a subroutine in your program does not adequately declare all of the memory it uses for arrays or character strings, it might be best for you to delay converting the program to an EPF until you have checked the program thoroughly and corrected any misdeclarations.

Programs That Call R-mode or S-mode Subroutines

A program that calls one or more subroutines written in R mode or S mode cannot be built as an EPF. Moreover, such a program cannot be built using SEG. (S mode is an archaic instruction set that predates R mode and that is still supported by Prime systems.)

Requiring Defined Initial Values for Registers

The static-mode runfile provides for initial values for the A, B, X, and K (Keys) registers. Some programs, particularly very old programs, continue to use these initial values as configuration or option selection information. For example, Prime's FTN and PMA translators use the initial values of the A and B registers to select both system-wide and user-specific compiler options, primarily for compatibility reasons.

EPFs do not support the notion of definable initial values for these registers, except for the K (Keys) register, which is defined by the ECB of the main entrypoint of the EPF. Therefore, a program that expects the initial values of the A, B, or X registers to be defined cannot be converted to an EPF until it is redesigned so that it no longer has that dependency.

If the first executable statement in a FORTRAN program is either CALL GETA or CALL GETL, or if the first instruction in a PMA program is STA, STX, or STL, then the program depends on the initial values of the A, B, or X registers. You must determine what information is being passed in the registers and redesign the program to acquire the information from a different source, such as command line options and external configuration information (such as a file), before you can convert the program to an EPF.

Programs That Depend on Automatic Initialization

SEG typically initializes static data (including common areas) to all zeros, unless otherwise specified by the program (via a DATA statement

in FORTRAN, an INITIAL attribute in PL1/G, and so on). BIND performs no such initialization, which means that the initial values for variables not initialized by the program are undefined. Therefore, a program that depends on the automatic initialization by SEG may work as a SEG-built program but fail when built with BIND. The proper solution is to modify your program so that it uses the appropriate language statements, attributes, or instructions to set initial values for all variables before using them.

CONVERSION PROCEDURES

The first step in converting a static-mode program to an EPF is to make sure that it does not fall into one of the categories of programs that cannot be converted, listed above.

If it does fall into one of these categories, you have three choices:

- Rewrite and/or recompile the program so that it can be converted.
- Replace the program with a new program, written so that it follows the EPF guidelines given in Chapter 4.
- Continue to use the existing program as a static-mode program.

The second step is to consider whether the program follows the guidelines for EPFs given in Chapter 4. If it does not follow these guidelines, you might want to modify the program so that it does follow them.

Once you are satisfied that your program is convertible to an EPF, you BIND the program.

REWRITING LINKING SEQUENCES

One question may remain. You may be looking at an existing program in which the program seems straight-forward enough, but the sequence of SEG subcommands used to link and load the program is far from simple. How do you convert this sequence of SEG commands into a sequence of BIND commands?

The following section provides information to help you make this conversion. It lists some of the SEG subcommands and either describes their BIND equivalents or explains why the action they perform is not needed with BIND. The section assumes that you have found the load sequence used to build the program in a command file or in a CPL program.

Invocation of SEG: There are five ways to invoke SEG. Two of them, SEG and SEG -LOAD, are similar to typing BIND. The SEG pathname form is replaced by RESUME pathname when working with EPFs. Instead of SEG pathname 1/1, use DBG pathname and then type VPSD. Instead of SEG 1/1, enter the VPSD command.

Use of SPLIT, MIX: A traditional SEG loading sequence that generates a RESUMEable runfile is:

```
SEG
SPLIT
MIX
CO ABS 4000
S/LOAD prog 0 4000 4000
D/LOAD subr1
D/LOAD subr2
    .
    .
    .
D/LIBRARY
RETURN
SHARE
prog
DELETE
QUIT
```

(The CO ABS 4000 subcommand is sometimes omitted.) With BIND, the following subcommand sequence accomplishes the equivalent result:

```
BIND
LOAD prog
LOAD subr1
LOAD subr2
   .
   .
   .
LIBRARY
FILE
```

If you wish a map, use the RESOLVE_DEFERRED_COMMON subcommand (abbreviated RDC) followed by the appropriate MAP subcommand of BIND just prior to issuing the FILE subcommand.

The SEG>CMDSEG Utility: The SEG>CMDSEG utility is used to generate a RESUMEable program that invokes a SEG runfile. Use of CMDSEG is totally unnecessary with an EPF; simply omit the step that runs CMDSEG, and copy the .RUN file generated by BIND to wherever the .SAVE file generated by CMDSEG was copied.

The DELETE Subcommand: The DELETE subcommand of SEG is often used to delete the SEG runfile version of a program after the creation of a static-mode runfile version that is RESUMEable by using the SHARE or SPLIT subcommands. In this case, it is not applicable with BIND, because the .RUN runfile generated by BIND is RESUMEable.

The MAP (MA) Subcommand: The MAP subcommand of SEG is available at both SEG level and at the VLOAD processor level.

At SEG level, the MAP subcommand may specify the SEG runfile. In BIND, you do this by issuing the command:

```
BIND -LOAD program.RUN -MAP [map-file] [map-option] -QUIT
```

This command replaces the following SEG sequence and its derivatives:

```
SEG
MAP program [map-file] [map-option]
QUIT
```

If the MAP subcommand does not specify the SEG runfile or if it is issued at the VLOAD processor level, then you issue it in a similar fashion in BIND, using the subcommand format shown in Chapter 8. If you have finished linking modules and libraries, you should issue the RESOLVE_DEFERRED_COMMON subcommand (abbreviated RDC) to cause BIND to allocate the common areas not already initialized. Otherwise, the common areas are listed as "deferred" in a BIND map, rather than being shown with their addresses.

The following table correlates SEG map-option numbers with BIND map-options, although the functionality is not precisely duplicated in each case:

SEG Map Option	BIND Map Option
0	(none needed)
1	-RANGES
2	-BASE
3	-UNDEFINED
4	(none needed)
6	-UNDEFINED
7	(none needed)
10	-NAMED_SYMBOL
11	(none needed)

In addition, in SEG, map files are written to file unit 13, and repeated uses of the MAP subcommand directing output to a single file causes information to be appended to the file by keeping the file open on unit 13. Because the MAP subcommand of BIND overwrites whatever file it is given as the output file, you must write each map to a separate file. To produce a single file containing all the maps, you can use the CONCAT command, described in the PRIMOS Commands Reference Guide.

The MODIFY (MO) Subcommand: The MODIFY subcommand of SEG places the user in a special MODIFY processor. Of the functions available in SEG's MODIFY processor, only one has a corresponding BIND function — the START (ST) subcommand. All of the other functions are not available (and probably not necessary) when you use BIND. BIND does not control the size or placement of the stack, deal with segments below '4000, or patch runfiles.

The START subcommand of the MODIFY processor changes the starting address of a SEG runfile by specifying the absolute segment number and octal address of the new main ECB. In BIND, you use the MAIN subcommand to specify the name of the new main entrypoint, as follows:

```
BIND
LOAD program.RUN
MAIN new-main-entrypoint
FILE [new-program.RUN]
```

The MODIFY subcommands are listed after the SEG subcommands and the VLOAD processor subcommands, at the end of this section.

The PARAMS (PA) Subcommand: The PARAMS subcommand displays information about the starting ECB address, stack placement, and initial register values for a SEG runfile. An EPF has no stack placement or initial register values. To display the starting ECB address, use the MAP subcommand of BIND.

The PSD (PS) Subcommand: The PSD subcommand invokes the VPSD debugging utility. BIND does not include a copy of VPSD because you cannot execute a program from BIND as you can from SEG. Use DBG to restore the EPF runfile and then use the PSD subcommand of DBG to examine the runfile.

The QUIT (Q) Subcommand: The QUIT subcommand saves the SEG runfile, returns to PRIMOS command level, and closes all open files. In BIND, QUIT never saves the EPF runfile; instead, it queries the user as to whether the user really wants to quit without saving the runfile. Also, the QUIT subcommand of BIND does not close all open files, because BIND does not make use of statically allocated open file units as does SEG.

The RESTORE (REST) Subcommand: The RESTORE subcommand restores a SEG runfile to memory. BIND never loads an EPF runfile into memory, nor does the PRIMOS RESTOR command restore an EPF to memory. To place an EPF in memory without executing it, either use DBG or write a program that calls the EPF$ subroutines described in the Advanced Programmer's Guide, Volume III: Command Evironment. If the RESTORE subcommand is being used to modify the EPF (for example, to relink the object file), use LOAD program.RUN followed by the appropriate RELOAD subcommands.

The RESUME (RE) Subcommand: The RESUME subcommand restores a SEG runfile to memory and either begins to execute it or enters VPSD if 1/1 was specified on the RESUME command line. Use the PRIMOS RESUME command to invoke an EPF runfile. Use DBG to restore an EPF runfile followed by the VPSD subcommand of DBG to examine the runfile using VPSD.

The SAVE (SA) Subcommand: At SEG level, the SAVE subcommand is the same as the MODIFY subcommand. See the description of the MODIFY subcommand in this section.

At the VLOAD processor level, SAVE writes the SEG runfile to disk, sets the stack location, and optionally allows the user to specify the initial values of the A, B, and X registers. In BIND, use the FILE

subcommand to both write the EPF runfile and exit to PRIMOS. The stack is not placed by BIND, and EPFs do not have initial values for the A, B, and X registers.

Normally, you would use the FILE subcommand in BIND to replace a SAVE subcommand followed by a QUIT subcommand in SEG. Delete SAVE subcommands that have no arguments. However, if you want to convert a program to an EPF and the program depends on the A, B, and X registers' having initial values, you must redesign that interface so that it does not depend on those registers.

The SHARE (SH) Subcommand: The SHARE subcommand prepares a SEG runfile for sharing by generating static-mode runfile images of segments below '4001. The images of segments below segment '4000, if any, are placed into shared memory at system coldstart, usually by the PRIMOS.COMI or C_PRMO file in CMDNC0. The image of segment '4000 serves as the RESUMEable program image that uses the shared memory portions of the program.

Because BIND produces EPF runfiles that are automatically shared by PRIMOS, there is no corresponding subcommand, and no activities need take place at system coldstart to allow an EPF runfile to be shared among users. However, if your application is particularly large and requires fast startup time, you might try comparing the performance of your application built as an EPF to its performance when built as a shared static-mode program before completely converting over. Some shared static-mode applications may realize performance losses when converted to EPFs, although this should be the exception rather than the rule.

The SINGLE (SI) Subcommand: Similar to the SHARE subcommand, the SINGLE subcommand prepares a large SEG runfile to be RESUMEable. SINGLE does so by generating a static-mode runfile image of a particular segment of the runfile. Because EPFs are directly RESUMEable, there is no corresponding command in BIND.

The TIME (TI) Subcommand: The TIME subcommand displays the date and time of the last modification to a SEG runfile. There is no BIND equivalent to this subcommand, but there is a PRIMOS command, LIST_EPF -EPF_DATA, that displays, among other information, the date and time an EPF was generated with BIND.

The VERSION (VE) Subcommand: The VERSION subcommand displays the version number of SEG. BIND displays its version number each time you invoke it. In addition, the PRIMOS command LIST_EPF -EPF_DATA displays the version number of BIND used to generate the EPF, in addition to the program version number if it is supplied by the user during the BIND session.

The VLOAD (VL) Subcommand: The VLOAD subcommand places the user in the VLOAD processor, which is where the loading of modules actually takes place. VLOAD also specifies the name of the SEG runfile to be generated or specifies the name of the SEG runfile to which new modules are to be appended.

When you use BIND, replace the sequence:

```
SEG
VLOAD pathname
```

with:

```
BIND pathname
```

Replace the sequence:

```
SEG
VLOAD * pathname
```

with:

```
BIND
LOAD pathname.RUN
```

Subcommands used in the VLOAD processor are listed next.

The ATTACH (A) Subcommand: The ATTACH subcommand attaches the user to another directory. This subcommand is obsolete in SEG and is not provided in BIND, because both SEG and BIND support pathnames in all cases.

The AUTOMATIC (AU) Subcommand: The AUTOMATIC subcommand allows the automatic generation of additional base areas during program loading. There is no corresponding BIND subcommand because BIND automatically provides a more sophisticated version of this feature.

The A/SYMBOL (A/SY) Subcommand: The A/SYMBOL subcommand defines a symbol in absolute memory and reserves space for it. In BIND, you use the SYMBOL subcommand with a slightly different syntax: you specify the complete address in segno/offset form, and you do not specify the type of segment for the symbol. Initialized common areas are not initialized during program loading if they are placed by using the

SYMBOL subcommand of BIND. See also the ALLOCATE subcommand of BIND for reserving space with no concern for location.

The COMMON (CO) Subcommand: The COMMON subcommand specifies the type of segment into which a COMMON block is loaded. No corresponding BIND subcommand exists because in BIND, COMMON blocks are always placed in DATA segments.

The D/ Prefix: The D/ prefix specifies that the load subcommand it prefixes is to be performed with the default parameters already established via use of the P/ or S/ prefixes. These parameters distinguish between segments (shared versus unshared). Because such parameters are not needed with BIND, there are no prefixes in BIND. Remove all D/ prefixes and all S/ or P/ parameter numbers, and substitute the -FORCE, -PAGE, or -FORCEPAGE options for the F/ and P/ prefixes.

The EXECUTE (EX) Subcommand: The EXECUTE subcommand saves and executes the SEG runfile. With EPFs, you issue the FILE subcommand of BIND to save the runfile, then you use the RESUME command of PRIMOS to execute the program.

The F/ Prefix: The F/ prefix specifies that all routines contained in the object file specified in the load subcommand it prefixes are to be forcibly loaded, thereby overriding RFL/SFL flags generated by the binary editor, EDB. To accomplish this in BIND, use the -FORCE option prior to the name of the object file.

The IL Subcommand: The IL subcommand, which is normally used when building shared subsystems, loads the impure FORTRAN library IFTNLB. There is no corresponding BIND subcommand, because BIND automatically builds shared programs.

The INITIALIZE (IN) Subcommand: The INITIALIZE subcommand initializes and restarts the VLOAD processor. To accomplish this with BIND, use the QUIT subcommand of BIND followed by a subsequent invocation of BIND.

The LIBRARY (LI) Subcommand: The LIBRARY subcommand loads libraries from UFD LIB and optionally specifies parameters or loading options via octal numbers and prefixes. In BIND, these options are specified by placing the -FORCE, -PAGE, or -FORCEPAGE option in front of a library pathname that is to be affected by that option. Octal number parameters may be discarded because BIND does not use them when linking libraries.

Note that a frequently used abbreviation of LIBRARY, LIB, does not work in BIND, which uses LI as the abbreviation. Change LIB to either LI or LIBRARY when you convert the build file for a program from SEG to BIND.

The LOAD (LO) Subcommand: The LOAD subcommand loads modules and optionally specifies parameters or loading options via octal numbers and prefixes. In BIND, these options are specified by placing the -FORCE, -PAGE, or -FORCEPAGE option in front of a module that is to be affected by that option. Octal number parameters may be discarded because BIND does not use them when linking modules.

The MAP (MA) Subcommand: See the description of the MAP subcommand earlier in this list, in the SEG-level subcommands.

The MIX (MI) Subcommand: The MIX subcommand allows the loading of linkage data and COMMON blocks in procedure segments primarily to allow the generation of a RESUMEable version of the program. There is no corresponding subcommand in BIND because EPFs are directly RESUMEable and cannot combine linkage data and COMMON blocks with pure procedure code. Omit this subcommand when you convert the build file.

The MV Subcommand: The MV subcommand moves portions of the SEG runfile from one memory location to another. There is no corresponding BIND subcommand. The MV subcommand is not a user command but is intended primarily for the creation of shared libraries by Prime. If your application uses MV to create a shared library, you may wish to convert the shared static-mode library to a library EPF. For information on library EPFs, see Chapter 10 of this guide and also the Advanced Programmer's Guide, Volume I: BIND and EPFs.

As far as what to do with the MV subcommand and other subcommands and PRIMOS commands used to generate, install, and initialize a static-mode library, most of it is discarded when you convert it to a library EPF. It is better to start from scratch by following the directions in the Advanced Programmer's Guide, Volume I: BIND and EPFs.

The NSCW (NS) Subcommand: The NSCW subcommand suppresses warning messages issued when a common area is defined by a loaded module to be a certain size, then redefined by a subsequently loaded module to be smaller. The BIND equivalent is the NO_COMMON_WARNING subcommand (abbreviated NCW). Because BIND allows subsequently linked modules to redefine uninitialized common areas to be larger, the NO_COMMON_WARNING subcommand inhibits warning messages for both smaller and larger redefinition of common areas.

Note

BIND allows the redefinition of a common area to be larger only if the common area is deferred. Deferral of common areas is the default in BIND. However, once you issue the RESOLVE_DEFERRED_COMMON subcommand, BIND no longer permits the redefinition of a common area to be larger. Similarly, if you link a module that initializes a common area, that common area cannot subsequently be redefined to be larger; otherwise, BIND rejects the attempt.

The OPERATOR (OP) Subcommand: The behavior of the OPERATOR subcommand may vary from revision to revision and has no equivalent BIND subcommand or option.

The P/ Prefix: The P/ prefix, together with a subcommand that loads a module, specifies that the module is to be loaded starting on a page boundary. In BIND, use the -PAGE option in front of the module or modules to be started on a page boundary when you use the LOAD or LIBRARY subcommands.

Note

The -PAGE option places only the procedure code portion of a module on a page boundary. However, the link frame, containing data, is not necessarily placed on a page boundary. With SEG, both the procedure frame and link frame are loaded on page boundaries, unless the PR or DA option is specified. PR specifies that only procedure code is to be loaded on a page boundary, and DA specifies that only linkage data is to be loaded on a page boundary.

Therefore, use of the P/ prefix along with the PR option in SEG is identical to use of the -PAGE option of BIND. Use of the P/ prefix with no option or with the DA option in SEG has no BIND equivalent, because BIND cannot place link frames on page boundaries.

In fact, BIND does place the link frame of a module linked with the -PAGE option on a page boundary relative to the start of the linkage data for the entire EPF. However, when the EPF is executed, the linkage data is not likely to start on a page boundary.

The PL Subcommand: The PL subcommand, normally used when building shared subsystems, loads the pure FORTRAN library PFTNLB and the system library SPLLIB. There is no corresponding BIND subcommand, because BIND automatically builds shared programs.

The QUIT (Q) Subcommand: The QUIT subcommand saves the SEG runfile and returns to PRIMOS command level. Use the FILE subcommand of BIND instead, because the QUIT subcommand of BIND does not save the EPF runfile.

The R/SYMBOL (R/SY) Subcommand: The R/SYMBOL subcommand defines a symbol and reserves space for it. ALLOCATE is the equivalent BIND subcommand. No relative segment number or segment type is specified with ALLOCATE.

The RETURN (RE) Subcommand: The RETURN subcommand saves the SEG runfile and returns to SEG subcommand level from the VLOAD processor level. There is no equivalent BIND subcommand, because BIND has only one subcommand level.

The RL Subcommand: The RL subcommand replaces binary modules in the current SEG runfile with new binary modules. The equivalent BIND subcommand is RELOAD (abbreviated RL). The starting address, procedure segment, and linkage segment are not specified with RELOAD. To enter BIND in order to replace one or more binary modules in an EPF, type:

```
BIND
LOAD EPF-program.RUN
```

You must specify the .RUN suffix, or else BIND first looks for the file EPF-program.BIN. After linking the EPF runfile in this fashion, issue one or more RELOAD subcommands. You may have to reissue one or more LIBRARY subcommands if the new versions of the binary modules call subroutines in libraries that the old versions did not call.

The S/ Prefix: The S/ prefix, combined with one of the VLOAD loading subcommands, specifies absolute segments into which the binary modules are to be loaded. Because the S/ prefix is normally used to build RESUMEable runfiles or shared subsystems, there is no equivalent BIND subcommand, prefix, or option.

The SAVE (SA) Subcommand: The SAVE subcommand saves the current SEG runfile. In BIND, the FILE subcommand both saves the EPF runfile and exits BIND, and is therefore more equivalent to the QUIT subcommand of SEG than to the SAVE subcommand.

The SAVE subcommand of SEG accepts as many as three arguments that assign initial values to the A, B, and X registers. Because EPF runfiles do not support the notion of initial register values, there is no equivalent functionality in BIND. If your program depends upon the initial values of the A, B, or X registers, you must redesign its interface before converting your program to an EPF.

The SCW (SC) Subcommand: The SCW subcommand reactivates the warning message issued when a common area is defined by a module to be a certain size, and subsequently redefined as smaller. The BIND equivalent is the COMMON_WARNING subcommand (abbreviated CW). See the description of the SEG subcommand NSCW, earlier in this section, for more information about the redefinition of the size of a common area.

The SETBASE (SE) Subcommand: The SETBASE subcommand creates a base area for address resolution linkages within a segment. There is no equivalent BIND subcommand or option, because BIND automatically creates base areas as it links modules.

The SPLIT (SP) Subcommand: The SPLIT subcommand breaks a segment into procedure and data portions, and also loads the RUNIT module and a stack overflow handler. There is no equivalent BIND subcommand because EPFs always place procedure and data in separate segments, because EPFs are directly RESUMEable, and because EPFs use the command processor stack (which has its own stack overflow handler).

The SS Subcommand: The SS subcommand protects a symbol from being deleted by the XPUNGE subcommand. There is no equivalent BIND subcommand because BIND does not provide the ability to delete all symbols (although you may use the CHANGE_SYMBOL_NAME subcommand to effectively remove specific symbols).

The STACK (ST) Subcommand: The STACK subcommand sets the minimum size of the stack. Because EPFs use the command processor stack when they execute, there is no equivalent BIND subcommand.

The SYMBOL (SY) Subcommand: The SYMBOL subcommand defines a symbol but does not reserve space for it. There are three forms of the SYMBOL subcommand:

Form 1: SYMBOL new-symbol-name old-symbol-name [offset]

Form 2: SYMBOL [new-symbol-name] segno [addr] [offset]

Form 3: SYMBOL [new-symbol-name] * [offset]

The equivalent BIND subcommand for Forms 1 and 2 is the SYMBOL subcommand. However, the SYMBOL subcommand of BIND does not accept the optional offset argument. SYMBOL may place a static symbol (for example, SY MY_COM 2037/0).

There is no equivalent BIND subcommand for Form 3.

The SZ Subcommand: The SZ subcommand controls the use of sector zero base areas in procedure segments. Because BIND automatically creates base areas in sector zero and in other areas as needed, there is no equivalent BIND subcommand.

The XPUNGE (XP) Subcommand: The XPUNGE subcommand deletes defined symbols as indicated by an argument, and optionally deletes base area information. There is no equivalent BIND subcommand. However, you may effectively delete specific defined symbols by using the CHANGE_SYMBOL_NAME subcommand of BIND to change a symbol's name.

The NEW (NE) Subcommand: The NEW subcommand of the MODIFY processor writes a partial copy of the SEG runfile to disk. There is no equivalent BIND subcommand. To build a template with BIND, issue the FILE subcommand at the desired point. This generates an EPF template with the .RUN suffix.

Then, to use the template when building a program, link (with the LOAD subcommand) the EPF template before linking any other modules. Make sure that you specify the complete name of the EPF template (including the .RUN suffix) to ensure that BIND links the EPF and not an object (.BIN) file with the same basename.

The PATCH (PA) Subcommand: The PATCH subcommand of the MODIFY processor writes a patch to disk. With BIND and EPFs, there is no corresponding function or subcommand.

The RETURN (RE) Subcommand: The RETURN subcommand of the MODIFY processor saves the SEG runfile and returns to SEG subcommand level. Because BIND has only one subcommand level, there is no equivalent BIND subcommand.

The SK Subcommand: The SK subcommand of the MODIFY processor specifies the stack size, location, and an extension stack segment. There is no equivalent BIND subcommand because EPFs use the command processor stack.

The START (ST) Subcommand: The START subcommand of the MODIFY processor specifies a new starting ECB for the SEG runfile. The equivalent BIND subcommand is MAIN, which accepts the name of the new main entrypoint, rather than its ECB address as a segment number/offset pair.

The WRITE (WR) Subcommand: The WRITE subcommand saves a portion of the SEG runfile to disk. There is no equivalent BIND subcommand. With BIND, you can link an EPF template, link in additional modules as desired, and then issue the FILE subcommand to write out the new EPF.

6

Libraries and EPF Libraries

Chapter 6 tells you what a library is, when a library is useful, and how to use a library. The chapter also discusses shared and unshared libraries and the two types of EPF libraries you create with BIND. For additional information on EPF libraries, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

WHAT IS A LIBRARY?

A library is a collection of separate routines bound together into one single file. Each library has a table of entrypoints for each of the contained routines. An entrypoint links a name with the actual location of the routine. Therefore, programs external to a library may call subroutines within the library at execution time. An example of a library is the FORTRAN library, which contains code for such functions as the SIN and COS routines. A library routine differs from a nonlibrary routine only in that the library routine is kept in a library.

WHEN IS A LIBRARY USEFUL?

When a procedure or function can be used by more than one program, it should be collected with other routines that can be used by those programs. The compiler for a specific language provides some of its support by calling routines in a specific library for that language.

Time is saved because library routines are compiled and prelinked independent of the program that uses them.

There are many operations that are useful in a variety of programming tasks. Among them are:

- System-level operations
- Commonly used language functions such as SIN, LOG, SQRT, MOD and I/O handling
- Application tasks such as string handling, file sorting, and system interrogation

Without libraries, you would have to write and debug your own routines for every programming operation. This could waste large amounts of programming time because you and your colleagues might independently create separate routines that perform identical functions.

Libraries provide an answer to this problem. When a routine needed by your program is kept in a library, you can include the library routine in your program, thereby eliminating the need to write and compile a routine of your own.

PRIMOS supplies you with a variety of standard libraries. These libraries and the routines they contain are described in the Subroutines Reference Guide.

In some cases, you may find that you need libraries that are not provided by Prime as standard software. Whenever a related set of routines are used in numerous programs, it is a good idea to keep them in a library.

HOW TO USE A LIBRARY

To include a library routine in a program:

- Reference the routine as needed in the program.
- Use the LIBRARY subcommand to name the library that contains the routine when you BIND the program (that is, when you link the library containing either the code for that routine or the reference to the entrypoint name for that routine).

The system does the rest automatically.

TYPES OF LIBRARIES

PRIMOS supports three types of libraries:

- Nonshared library. A nonshared library is a binary module (a .BIN file) that contains the code for the routines you link into your program.
- Static shared library. A static shared library keeps its code in one or more shared segments in the range '2000 through '2777, and uses an area of memory in segment '6001 or '6006. The static shared library is loaded into memory when the system is started up. To use this library, you link a binary module (a .BIN file) that contains a list of named entrypoints for that library. Therefore, a static shared library does not become part of your program.
- EPF library. An EPF library is an EPF runfile (a .RUN file) that contains the code for the routines your program uses. To use this library, you link a binary module (a .BIN file) that contains a list of named entrypoints for that library. This type of library dynamically becomes part of your program at execution time, but is not part of your program when your program is not running.

To PRIMOS, the three types of libraries are quite different. The two major differences are (1) that a routine from a nonshared library is physically part of a runfile that uses it, whereas a routine from a static shared library or an EPF library is a separate file; and (2) that routines in a static shared library are always available in memory, whereas routines in an EPF library can be accessed only when that library is part of the user's entrypoint search list.

Nonshared Libraries

A nonshared library is a sequence of compiled routines that are kept together in one .BIN file.

When you issue a LIBRARY subcommand that names a nonshared library, the library you name is scanned. Any routine in the library that is referenced in the program being linked is physically copied into your runfile at execution time. (If the Set Force Load (SFL) flag is in effect, then all routines are linked.) The resulting runfile looks just as it would have if all such routines had been compiled by you and linked with a LOAD subcommand.

Static Shared Libraries

With static shared libraries, the code for a routine is in a particular location and it is referenced by using the entrypoint name for that

routine. Only one copy of a static shared routine exists on a system, and it remains at all times within its shared library. All programs that use a static shared routine execute this master copy, rather than making copies of their own.

EPF Libraries

With EPF libraries, the code for a routine resides in a file containing a library EPF runfile. The routine is referenced by using its entrypoint name.

Prime supplies several library EPFs in the directory LIBRARIES* that are used by most programs. Users may build their own library EPFs in their own directories. They may allow other users to use routines in these library EPFs by granting the other users sufficient access to the library EPF files.

Whether a library EPF resides in the LIBRARIES* UFD or in a user's own directory, it is automatically shared by PRIMOS if more than one user is using it at a time.

LINKING A PROGRAM TO A LIBRARY ROUTINE BY NAME

You cannot create a direct connection between a program and the location of a routine in a static shared library or an EPF library at program link time. The reason is that such a connection requires the address of the routine to be included in the program runfile. The address of the routine, however, may change from one use of the library to the next (particularly for EPF libraries).

Instead, PRIMOS dynamically connects a program to a routine in a static shared library or an EPF library at program runtime by using dynamic links.

How Dynamic Linking Works

A dynamic link is a special kind of pointer that contains the information needed by PRIMOS to connect an executing program to a library routine. When a static shared library or an EPF library is linked into a runfile, the actual code in the library is not linked. Instead, a dynamic link to each needed entrypoint in the library is put into the program in place of the actual code for the routine. When PRIMOS encounters this dynamic link while the program is running, PRIMOS changes the link into a pointer to the actual code in the library that corresponds to the entrypoint specified by the name in the dynamic link.

EPF LIBRARIES

EPF libraries have all of the same properties as program EPFs:

- They are allocated memory by PRIMOS at runtime.
- They execute in any private dynamic segment.
- Their program image is automatically shared if it is used by more than one user at a time.
- Their linkage is automatically allocated and initialized by PRIMOS at runtime.

You can create your own personal libraries as EPF libraries. These libraries have the same properties as EPF libraries supplied by Prime in the LIBRARIES* UFD. Linkage for an EPF library is usually initialized the first time a link is made to an entrypoint within the library.

Unlike a program EPF, a library EPF cannot be RESUMEd. A library EPF is dynamically linked to a program at program runtime. This linkage occurs when the running program makes a reference to an entrypoint within the library. The dynamic linking mechanism in PRIMOS automatically detects that the entrypoint is within an EPF library and executes the library as an EPF. PRIMOS maps the library EPF into memory and connects the running program to the desired entrypoint by turning the dynamic link into an actual memory pointer. PRIMOS then resumes program execution, retrying the reference to the entrypoint. This time the desired subroutine is executed.

There are two different types of EPF libraries:

- Program class
- Process class

The two EPF library types are differentiated by their initialization requirements. The following sections discuss the two library classes and the rules involved in linking to libraries. How to create the two types is discussed in the section HOW TO CREATE YOUR OWN EPF LIBRARY below.

Program-class Libraries

A program-class library acts as if it is physically part of any runfile that invokes it.

A program-class library is given a different linkage area for every program that uses the library. The area is created the first time any routine from the library is invoked, and it is retained until the invoking program returns to PRIMOS, at which time the storage area is

deallocated. There is a separate storage area for every program that is using a program-class library at a given time.

Say, for example, that you have two programs that will use the same library. If the library is an EPF program-class library, it is given a new linkage area for each of the two programs. If the library is a static shared library, the library is reinitialized when the second program links to it. Because there is only one linkage area for a static shared library, this reinitialization corrupts the library data for the first program.

Process-class Libraries

A process-class library is a special-purpose library that is user-specific rather than program-specific. This type of EPF library is called process-class because PRIMOS treats each logged-in user as a single process.

PRIMOS allocates a new linkage area to a process-class library the first time a program invokes any routine in the library during a terminal session. All programs that run subsequently at any command level use that same linkage area when they use that library. The process-class library linkage area is retained until you log out, until you reinitialize your command environment, or until you explicitly remove the library.

Process-class libraries are useful for sets of entrypoints that need their linkage initialized only once. This group includes routines whose actions are determined solely by their input arguments, any constant data needed, and any local variables (variables that are kept on the stack).

Sometimes none of the routines in a library use any of the allocated linkage area. Such a library is most efficient if it is process-class, because PRIMOS omits nearly all checking of linkage area requirements when a process-class library routine is invoked.

Interaction of the Classes of Library

A library routine may invoke other library routines. When one routine calls another in the same library, no special action is needed. When one routine calls another in some other library, PRIMOS may have to create a linkage area for the second routine.

When the two libraries are of the same class, no problem arises, because PRIMOS handles the linkage areas for both libraries in the same way. When the two libraries are of different classes, problems can arise because different types of libraries have different linkage initialization requirements. For example, links from a process-class library to a program-class library routine are not allowed. If you try

to link to a program-class library routine from a process-class library, PRIMOS issues an error message. The reason for this restriction is that a process-class library allocates a new linkage area only once, whereas a program-class library allocates a new linkage area every time the program is invoked.

Links from a program-class library to a process-class library are permissible. However, if both libraries are permitted to use each other's routines indiscriminately, PRIMOS cannot tell when to create a new linkage area for a library, because linkage areas are allocated only as a result of encountering a dynamic link, and dynamic links from a process-class library to a program-class library might already be resolved even when a new program calls the process-class library.

Whenever a static-mode library is in use, linking to it is valid only within the same program invocation that originally invoked the static-mode library. If a new program (that was invoked at a higher command level) invokes a static-mode library that is in use by another program run by the user, the invocation is allowed, but any other programs that have used the static-mode library are marked as "not restartable" by PRIMOS because the data area for the static-mode library has been changed.

HOW TO CREATE YOUR OWN EPF LIBRARY

You create an EPF library by using BIND. To create an EPF library runfile, follow these three steps:

1. Designate the class of the library.
2. Link the binaries, designating the entrypoints of the library.
3. Save the library.

The following sections discuss these steps.

Step 1 — Designate the Class of the Library: To designate the class a library is to have, use the LIBMODE subcommand. The LIBMODE subcommand has the following format:

```
LIBMODE  { -PROCESS }
         { -PROGRAM }
```

Choose one of the following modes for the LIBMODE subcommand:

- -PROCESS to indicate a process-class library
- -PROGRAM to indicate a program-class library

A process-class library is initialized only once for each process that references it. A program-class library is initialized once for each program invocation that references it.

The following example shows the use of the LIBMODE subcommand to designate a program-class library:

```
LIBMODE -PROGRAM
```

Step 2 -- Link the Binaries and Designate the Entrypoints: The runfile of an EPF library contains a table of entrypoints. PRIMOS uses this table to determine whether the runfile contains a routine referenced in a dynamic link. A routine in an EPF runfile is not accessible for dynamic linking unless its name appears in the entrypoints table for that library EPF.

To designate a routine in a library EPF as an entrypoint for that library, use the ENTRYNAME subcommand. The ENTRYNAME subcommand has the following format:

```
            ( -ALL          )
ENTRYNAME   { list-of-names }
            ( -NONE         )
```

This subcommand defines the names you give in list-of-names as entrypoints into the library EPF runfile you are currently binding. (You must link a routine before you can specify its name in list-of-names.) Alternatively, you may use the -ALL option to cause all successively linked subroutines to be made entrypoints; after linking your modules, use the -NONE option to prevent subsequently linked subroutines, such as those linked via the LIBRARY subcommand, from becoming entrypoints.

For example:

```
ENTRYNAME INTEREST DIVIDEND
```

This specifies that list of entrypoints to the library EPF being built includes INTEREST and DIVIDEND.

Another example:

```
ENTRYNAME -ALL
LOAD MYSUBS
ENTRYNAME -NONE
LIBRARY PL1GLB
LIBRARY
```

This specifies that all subroutines in the module MYSUBS should be in the list of entrypoints to the library EPF being built. Subroutines in the PL1/G library and in the standard system libraries, however, are not included in the list of entrypoints to that library EPF, because they are already entrypoints in other library EPFs supplied by Prime.

Step 3 -- Save the Library: After you link a library EPF, save it as you would an other EPF, by issuing the FILE subcommand:

```
FILE [EPF-library-name]
```

The library EPF runfile is now in your directory with a .RUN suffix. If you do not specify EPF-library-name, BIND adds the suffix .RUN to the first library routine you link with the LOAD subcommand.

HOW TO USE A LIBRARY EPF

To use a library EPF, you must do the following:

1. Create an entrypoint search list file to tell PRIMOS where on the system your library can be found and where in the system hierarchy you want it placed for your use.

2. Use the SET_SEARCH_RULES command to tell PRIMOS that you want to use your own entrypoint search list rather than the system search list.

3. When you build a program that uses the library EPF, use the DYNT subcommand of BIND to specify which entrypoints in your library the program will use.

The remainder of this chapter discusses these subjects. For more detailed discussion, see the *Advanced Programmer's Guide, Volume I: BIND and EPFs*. For more information on the SET_SEARCH_RULES command, see chapter 9.

Creating an Entrypoint Search List File

An entrypoint search list file is a text file created with an editor such as ED or EMACS. Create an entrypoint search list file in the following manner:

1. Enter a line containing the keyword -SYSTEM.

2. Give the library EPF pathnames, one per line, of the libraries you have created.

3. File the search list. The filename must have the .ENTRY$.SR suffix.

For example:

```
OK, ED
INPUT
-SYSTEM
MYUFD>LIBRARIES>TERMINAL_LIBRARY
MYUFD>LIBRARIES>PAYROLL_LIBRARY
(CR)
EDIT
FILE MYUFD>MYLIB.ENTRY$.SR
OK,
```

Enabling an Entrypoint Search List File

Now, issue the SET_SEARCH_RULES command (abbreviated SSR) to tell PRIMOS about the search list you have created. This command has the following format:

```
SET_SEARCH_RULES pathname
```

You do not have to specify the .SR suffix in pathname, but you must specify the .ENTRY$ suffix, or else PRIMOS does not recognize the search list as being for entrypoints.

For example, to use the entrypoint search list created above, type:

```
OK, SET_SEARCH_RULES MYUFD>MYLIB.ENTRY$
```

PRIMOS automatically inserts the system default rules at the top of your search list (but it does not modify the on-disk copy of your search list file).

You also may have your own libraries searched before the system default entrypoint search list. To do this, place your library entries into the entrypoint search list file before the -SYSTEM keyword, then use SET_SEARCH_RULES with the -NO_SYSTEM option. In this case, the command line from the above example becomes

```
OK, SET_SEARCH_RULES MYUFD>MYLIB.ENTRY$ -NO_SYSTEM
```

For more information setting search lists, see the Advanced Programmer's Guide, Volume II: File System.

Disabling an Entrypoint Search List File

To disable your own entrypoint search list file and reinstall the system entrypoint search list, use the SET_SEARCH_RULES command as follows:

```
SET_SEARCH_RULES -DEFAULT ENTRY$
```

Permanently Enabling Your Entrypoint Search List File

When you issue the SET_SEARCH_RULES command, your search list is set until one of the following occurs: you log out, you initialize your command environment, or PRIMOS automatically initializes your command environment as a result of an error condition. Your search list is then reset to the system default search list.

To permanently enable your entrypoint search list file, place the appropriate SET_SEARCH_RULES command at the top of your login file (such as LOGIN.CPL) in your origin directory. If you do not have a login file, create one as follows:

```
OK, ORIGIN
OK, ED
INPUT
SET_SEARCH_RULES MYUFD>MYLIB.ENTRY$
(CR)
EDIT
FILE LOGIN.CPL
OK,
```

To change your search list after you have set up your login file in this manner, you should modify your login file or the search list itself (or both), and then use the INITIALIZE_COMMAND_ENVIRONMENT command (abbreviated ICE). This procedure prevents any problems that might result from having a program use two versions of your entrypoint search list. You should use this procedure rather using the SET_SEARCH_RULES command to cause the changes to take effect.

Use of Private Entrypoint Search Lists With Phantoms or Batch Jobs

When you spawn a phantom (via the PHANTOM command) or submit a batch job (via the JOB command), the process uses the system default search list, SYSTEM>ENTRY$.SR, whether or not you have defined your own search list and whether or not your login file defines one. (This default situation may change in future revisions of PRIMOS.)

Therefore, if you wish a spawned phantom or submitted batch job to use your search list, you must place an appropriate SET_SEARCH_RULES command near the top of the command file or CPL program used to start the phantom or job.

Using the DYNT Subcommand of BIND

When you build a program that will use a library EPF that you have built, you can use the DYNT subcommand to define the entrypoints of that library EPF. (You need to define only the entrypoints used by the program being built.)

For example, if you are building a program that calls the entrypoints CLEAR_SCREEN, MOVE_CURSOR, and HIGHLIGHT_FIELD, you issue the following DYNT subcommand during the BIND session:

```
DYNT CLEAR_SCREEN MOVE_CURSOR HIGHLIGHT_FIELD
```

The placement of the DYNT subcommand in the BIND session is not important, although it is usually placed just before the MAP or FILE subcommand. For long lists of subroutines, you may issue more than one DYNT subcommand.

An alternative to the DYNT subcommand, explained in the *Advanced Programmer's Guide, Volume I: BIND and EPFs*, involves building an object file that is linked using the LIBRARY subcommand as with Prime-supplied libraries. This alternative, although more complicated, is recommended for libraries that will be used by more than one programmer.

7

Troubleshooting

PROBLEMS YOU MAY RUN INTO

This chapter discusses some of the problems you may encounter with EPFs. The following topics are covered:

- Running out of individual resources
- Running out of system resources
- Problems with in-use EPFs

RUNNING OUT OF INDIVIDUAL USER RESOURCES

When the System Administrator creates your login ID, he or she also sets up your command environment limits. These limits are:

- Maximum number of command levels for suspending program applications (command environment depth)
- Maximum number of simultaneous program invocations per command level (command environment breadth)
- Number of static segments that can be allocated in your private address space
- Number of dynamic segments that can be allocated in your private address space

Command Environment Depth

EPFs allow you to invoke a program, suspend it (by typing CONTROL-P), and invoke another program, suspend that and resume the first one without losing data in either program, as explained in Chapter 4. Each time you suspend a program, you are placed at another command level so that you can resume another program. The number of levels you can have is limited either by the number the Project Administrator enters in your user profile or by the number of levels the System Administrator has set up as the project limits.

To find out what your current command level is, use the RDY command without options. This command displays the OK prompt followed by the clock time, CPU and I/O time used since the last prompt, your current command level if greater than 1, and a plus sign if the level is static mode. In the following example, the RDY command is issued after using CONTROL-P twice:

```
OK, RDY                                 /* Start at command level 1.
OK 13:37:01  1.290  0.303
OK,                                     /* Use CONTROL-P twice.
QUIT.
OK,
QUIT.
OK, RDY                                 /* Now at command level 3.
OK 13:37:11  0.154  0.000  level 3
OK,
```

During program development, you may wish to use the RDY -LONG command, which enables the long form (described above) of the system OK, and ER! prompts.

Command Environment Breadth

Programs can call other programs without changing command level, as explained in Chapter 4 and as described in Chapter 10. The number of programs that can be called from a program is limited by the number of live program invocations per level that the Project Administrator specifies. (The Project Administrator may have entered a specific limit in your user profile or project profile, or the System Administrator may have set a project limit that is the same for all users.)

Segments

When you invoke a program, PRIMOS allocates one or more segments in your address space, where that program will run. PRIMOS allocates static or dynamic segments, depending on the type of program to be run.

Static-mode programs created by LOAD and SEG are loaded into predefined static segments. They can be overwritten by any other static-mode program that you invoke.

Programs created by BIND are called dynamic because they are loaded into any unused dynamic segments in your private address space. They can, therefore, co-exist with other dynamic programs. Library EPFs also use dynamic segments. Each separate library EPF is allocated at least one procedure segment.

The Project Administrator determines the number of static and dynamic segments each user is allotted, either individually or on a project-wide basis. The System Administrator may override project-wide limits with system-wide limits.

Mini-command Level

If you exceed the number of command levels allotted to you, you are placed at mini-command level. The following example shows the output you see if you exceed your command-level depth and reach mini-command level.

```
You have exceeded your maximum number of command levels.

You are now at mini-command level.  Only the commands shown
below are available.  Of these, RLS -ALL should return you to
command level 1.  If it does not, type ICE.  If this problem
recurs, contact your System Administrator.

Valid mini-commands are:

Abbrev  Full name             Abbrev  Full name
------  --------------------- ------  ------------------------------
C       CLOSE                 COMO    COMOUTPUT
DMSTK   DUMP_STACK            ICE     INITIALIZE_COMMAND_ENVIRONMENT
LE      LIST_EPF              LL      LIST_LIMITS
LMC     LIST_MINI_COMMANDS    LS      LIST_SEGMENT
        LOGIN                 LO      LOGOUT
P       PM                    PR      PRERR
        RDY                   REN     REENTER
RLS     RELEASE_LEVEL         REMEPF  REMOVE_EPF
S       START

OK,
```

At mini-command level the only commands you can execute are those listed in the display above. Your personal abbreviations are not available. INITIALIZE_COMMAND_ENVIRONMENT, LIST_MINI_COMMANDS, LIST_EPF, LIST_LIMITS, LIST_SEGMENT, and REMOVE_EPF are described fully in Chapter 8, the EPF Commands Dictionary. See the PRIMOS Commands Reference Guide for a description of the others.

Choosing the Right Mini-command: The following table lists some of the actions you may wish to take, and the command you will need to use.

Action	Command
Keep a record of your session.	COMOUTPUT
Find out what your limits are.	LIST_LIMITS
Find out what EPFs you are using.	LIST_EPF
Find out what segments you are using.	LIST_SEGMENTS
Find out what commands are available.	LIST_MINI_COMMANDS

The following are actions you can take to get out of mini-command level.

Action	Command
Remove EPFs from your address space.	REMOVE_EPF
Free unneeded dynamic segments.	REMOVE_EPF
Release unneeded command levels.	RELEASE_LEVEL
Return to a suspended program.	START
Return to a suspended subsystem.	START
	REENTER

REMOVE_EPF alone does not take you out of mini-command level. To do so, you must follow REMOVE_EPF with REENTER, RELEASE_LEVEL, or START.

The following table lists problems you may be having, and the commands that help you to solve the problem.

Problem	Command
Damaged command environment	ICE
Exceeded depth	RELEASE_LEVEL
Program error	DUMP_STACK
	PM
Used bad version of an EPF	REMOVE_EPF
Quit from program by mistake	START
Quit from a subsystem by mistake	START
	REENTER
Unable to delete a file	CLOSE
Unable to RESUME another EPF	REMOVE_EPF

Exceeding Command Depth: If you have used up the number of levels allocated to you, you can use RELEASE_LEVEL once for each level you can get rid of. (RELEASE_LEVEL -ALL frees all levels.) Or, you can return to the program you were previously executing, by using either REENTER or START. See the PRIMOS Commands Reference Guide to determine which is appropriate.

Exceeding Command Breadth: If your application exceeds command level breadth, the error code E$ECEB is returned to your program. To minimize this problem, have the program call RD$CED, as documented in the Subroutines Reference Guide, to return the current value of the command environment breadth. The maximum value may be obtained via the routine CE$BRD.

Exceeding Dynamic Segments: In most cases, you should not run out of dynamic segments while you are using the system. However, if your System or Project Administrator has set your limits to the minimum (16), it is possible to exceed your dynamic segment limit.

If the operating system tries to allocate dynamic data areas on your behalf and runs out of segments in your address space, the message:

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

appears on the terminal and your environment is reinitialized. If this occurred because you still had several segments allocated to suspended programs, you are probably now able to use the command (or run the program) that caused your environment to be reinitialized. If, on the other hand, you had no other segments allocated, you should ask your Project or System Administrator to allocate more dynamic segments for you. If an Administrator changes your allocation, you need to log in again for the new allocation to take effect.

An attempt to exceed the allowed number of dynamic segments may result in the message:

```
Not enough segments.  command-name (std$cp)
```

where command-name is the name of the command that overstepped the limit. If you already have several segments allotted to other programs, use RELEASE_LEVEL -ALL to release these programs, then try again. If this fails, your System Administrator or Project Administrator has to increase your dynamic segment limit for you to be able to use this command. Also, if your site uses library EPFs extensively, and all of these are included in the default entrypoint search list ENTRY$.SR, you may run out of dynamic segments. In this case, use private search lists for any search rules that are not used extensively at your site.

If the program you are running fails, you may receive the messages:

```
STORAGE raised in PROGRAM_NAME at nnnn
  (insufficient space for ALLOCATE)

ERROR raised in PROGRAM_NAME at nnnn
  (no on-unit for STORAGE)
```

One of three things may be happening: If you already have several segments mapped to your address space for other EPFs, use REMOVE_EPF to free up more segments, and try again. If the error recurs, use LIST_LIMITS to determine how many dynamic segments you are allotted. If it is a small number (less than 40), ask your Project or System Administrator to allow you more dynamic segments. If you have sufficient segments allotted to you, the program itself may have a problem.

Exceeding Static Segments: If some commands or utilities do not seem to work and you are not holding segments over from a suspended program, you probably need additional static segments.

The following message may be displayed in this case:

```
Error: condition "ILLEGAL_SEGNO$" raised at 4000(3)/16722.
(Referencing 4062(3)/1654).
ER!
```

In this example, the static segment being referenced is segment 4062. It is probably a dynamic segment, which cannot be accessed as a static segment. Your System Administrator or Project Administrator must increase your static segment limit for you to be able to use these commands or utilities.

RUNNING OUT OF SYSTEM RESOURCES

Occasionally you may not have overstepped your own allocation of segments, and yet you may still be unable to acquire a segment. What has probably happened is that the system itself has run out of available segments for EPF storage allocation. This problem should occur only when a system is heavily loaded. Use REMOVE_EPF to free your own unneeded EPFs and try again. If this does not free enough space, you may have to wait until other users have freed some space.

If the system is running out of segments, the error code returned to the program is E$NMTS (No More Temp Segments). The error code returned to the program if the system is running out of VMFA (Virtual Memory File Access) segments is E$NMVS (No more VMFA segments).

If the system runs out of segments frequently, your System Administrator may be able to increase either the number of VMFA segments available or the total number of segments available to all users.

Problems With In-use EPFs

If an EPF is in use, BIND and COPY automatically change the .RUN suffix of the in-use file from .RUN to .RPn (where n is a digit that ranges 0 through 9, inclusive) so that it can use the .RUN suffix for the new version of the EPF. If files already exist with all of the possible suffixes (.RP0 through .RP9), BIND and COPY check each .RPn file to find one that is not in use. If one is found, you are asked whether it should be deleted as in the following example.

```
OK, COPY LD.RUN CMDNC0>LD.RUN
EPF file "LD.RUN" already exists, do you wish to replace it? YES
ok to delete EPF file LD.RP0? YES
New version of EPF file LD.RUN now in place.
Old version of active EPF file now named LD.RP0.
OK,
```

You cannot delete an EPF that another user currently has mapped to his or her address space. If you attempt this, you receive the error message:

```
EPF file not active for this user. Unable to remove file NAME. (remepf$)
ER!
```

where NAME is the filename of the EPF you are trying to delete. To delete the EPF, you must wait until the it is no longer in use.

Problems With BIND

Because BIND has a built-in help facility, you do not have to exit from BIND to get help. The following example illustrates its use.

```
OK, BIND
[BIND rev 19.4]
: LOAD EXTLOG
: HELP LIBRARY
LIbrary <list of options and pathnames>
     will bind the files in the list to the current EPF.
     The options are the same as those for LOad.
: LI VAPPLB
: LI
BIND COMPLETE
: FILE
OK,
```

If you discover that a program linked with BIND does not work, you must correct the problem and then relink the program. If the program aborts or if you suspend it, you are placed at a lower (numerically higher) command level. You can then use the DUMP_STACK command and other debugging aids to trace the stack and examine the program's data areas.

If you relink a program that is mapped to another user's address space, BIND renames the old version of the program, as shown in the following example.

```
OK, BIND
[BIND rev 19.4]
: LOAD CIRCLE
: LIBRARY
BIND COMPLETE
: FILE
The file is in use.  The old file is now called CIRCLE.RPO.
OK,
```

Appendix A contains a comprehensive list of BIND error messages.

Problems With Libraries and Search Rules

Static-mode Libraries: A static-mode library is reinitialized each time it is invoked by a new program. This means that if a program calls a static-mode library, then calls another program which also calls the same static-mode library, PRIMOS refuses to allow the second

program to call the in-use static-mode library; instead, it produces an error message such as:

```
Error: condition "LINKAGE_ERROR$" raised at 4331(3)/1004.
Attempt to link to in-use static-mode library entrypoint "SPOOL$".
```

If, after running the first program, you quit to a new command level and attempt to run the second program, PRIMOS reinitializes the static-mode library and allows the second program to be run; it assumes that you do not wish to continue executing the first program. An attempt to REENTER or START the first program fails with the following message:

```
Attempt to proceed to overwritten program image. (listen_)
ER!
```

Search Rules: You should not run into problems if you are using the system-defined search rules. (If you do encounter problems, contact your System Administrator.) However, if you have set up your own search rules, unpredictable runtime errors may result if there are entrypoint-naming conflicts between libraries.

If you have set your own search rules, and discover that your program is behaving strangely, try issuing the SET_SEARCH_RULES command with the option -DEFAULT ENTRY$. This command restores the system-defined search rules. Try running your program again.

You may have to modify the login file in your origin directory if you cannot correct the problem.

Use NSED rather than ED or EMACS to edit the file:

```
OK, ORIGIN
OK, NSED LOGIN.CPL
EDIT
```

(NSED does not use the dynamic linking mechanism, and hence does not use the entrypoint search list.) Now, examine and possibly modify your login file as appropriate. If you modify your LOGIN.CPL file, use the INITIALIZE_COMMAND_ENVIRONMENT command (abbreviated ICE) to get it to take effect.

Using DBG

If running your program produces strange results, you may wish to compile the program in -DEBUG mode and step through the routine. After you find the problem, you can correct it and then recompile and relink the program.

Before you use DBG to examine an EPF, you should use the INITIALIZE_COMMAND_ENVIRONMENT command, described in Chapter 9, so that you start with a clean slate.

Using DBG on an EPF is no different from using DBG on other programs. To use DBG on a library EPF, compile it with the -DEBUG option and BIND its modules as a program EPF. When you are satisfied that it is working properly, BIND it again as a library EPF. For further information on this procedure, see the Advanced Programmer's Guide, Volume I: BIND and EPFs.

PART III

Reference

8
BIND Subcommands Dictionary

The basic subcommands given in Chapter 2 are sufficient for most of your linking needs. There are times, however, when you may want or need to use the more advanced features of BIND. For example, you may need to use these advanced features in order to:

- Increase the size of a COMMON block
- Inhibit the check for redefinition of COMMON blocks
- Create dynamic links to library routines
- Use LIBRARY options to override default loading
- Define symbol names

This chapter lists alphabetically all subcommands and options available with BIND. The chapter is divided into two sections:

1. BIND subcommands that govern the linking process (See Table 8-1 for a summary of these subcommands.)
2. BIND subcommands that govern command processing when the EPF is invoked (See Table 8-2 for a summary of these subcommands.)

Table 8-1
Summary of BIND Linking Subcommands

Command	Abbreviation	Function
ALLOCATE	ALLOC	Allocates storage for a symbol
COMMON_SYMBOL_NAME	CSN	Changes the name of a symbol
COMMENT		Inserts comment line in the EPF
COMMON_WARNING	CW	Checks size of COMMON block definition
COMPRESS		Gives compressed version of EPF
DYNT		Creates dynamic links
ENTRYNAME	EN	Defines entrypoints
FILE		Completes processing EPF
HELP		Gives help while in BIND
INITIALIZE_DATA	IDATA	Initializes data segments
LIBMODE	LM	Defines library EPF type
LIBRARY	LI	Links libraries
LOAD	LO	Links object files
MAIN		Designates main procedure
MAP	MA	Creates memory map of the EPF
NO_COMMON_WARNING	NCW	Inhibits checking for COMMON block redefinition
QUIT	Q	Terminates BIND session
RELOAD	RL	Replaces a binary file
RESOLVE_DEFERRED_COMMON	RDC	Forces space allocation for deferred COMMON blocks
SEARCH_RULE_VERIFY	SRVFY	Provides full pathnames of files being loaded
SYMBOL	SY	Defines a symbol
VERSION		Inserts version map

Table 8-2
EPF Command Line Subcommands
[Asterisks (*) indicate defaults.]

	Command	Abbreviation	Function
*	ITERATION	ITR	Performs iteration
*	NAMEGENPOS	NGP	Performs name generation
	NO_GENERATION	NG	Does not perform name generation
	NO_ITERATION	NITR	Does not perform iteration
	NO_TREEWALK	NTW	Does not perform treewalking
	NO_WILDCARD	NWC	Does not perform wildcard expansion
*	TREEWALK	TW	Performs treewalking
*	WILDCARD	WC	Performs wildcard expansion

BIND

As explained in Chapter 2, you can create an EPF, using BIND in one of two ways:

- You can run BIND at the PRIMOS command line. All control arguments that you issue correspond to internal BIND subcommands. You must precede each argument with a hyphen.
- You can run BIND interactively by invoking subcommands of BIND in response to the colon (:) prompt.

BIND automatically saves the EPF in your directory and gives the EPF the default name EPF-filename.RUN. If you do not specify EPF-filename, BIND adds the suffix .RUN to the first object filename that you link and saves the runfile in your directory. If you specify EPF-filename when you use the FILE subcommand, BIND automatically adds a .RUN suffix to the name you specify.

LINKING COMMANDS

▶ ALLOCATE symbol-name space

Abbreviation: ALLOC

Purpose: Use this subcommand to create and set the size of an external static data area, or COMMON block, that is part of a standard library routine. A standard library routine might contain a COMMON area that is integral to the routine's function, but which is too small for some desired application of the routine. For example, the buffer that holds the data transferred by an I/O routine might be a COMMON area. This area, though large enough for any expected use of the routine, might be too small for some application in which unusually large records must be read or written.

The ALLOCATE subcommand allocates a larger amount of space for the restricting COMMON area. Before the routine that uses the area is linked, name the area in an ALLOCATE subcommand, and give the desired size. The ALLOCATE subcommand allocates storage for symbol-name (the name of an external static data area) in one or more linkage segments. symbol-name must not exist before you use this subcommand. space is the number of 16-bit halfwords to allocate for symbol-name and may be over a segment (65536 halfwords) in size.

▶ CHANGE_SYMBOL_NAME old-name new-name

Abbreviation: CSN

Purpose: Use this subcommand to change the name of an existing symbol name. old-name is the name of an already loaded routine. new-name is the name of the new routine you wish referenced by calls that previously have been linked to old-name. new-name must not exist before you use this subcommand. Any modules linked after this subcommand is issued that reference old-name will not be changed to reference new-name. This subcommand affects only existing references, not future ones.

For example:

```
: LOAD PROG1.FTN                     /* Load a FORTRAN module
: LIBRARY                            /* Load the FORTRAN libraries
: CHANGE_SYMBOL_NAME EXIT GONE$      /* Change name of EXIT to GONE$
: LOAD PROG2.CC                      /* Load a C module
: LIBRARY CCLIB                      /* Load the C libraries
: LIBRARY                            /* Load standard system libraries
```

The C program can now use its own version of the EXIT routine, whereas the FORTRAN program uses the system version (a dynamic link to the PRIMOS EXIT routine). To reference a symbol by its old name, change it back to its former name before linking the module that references it.

▶ COMMENT [text-string]

Purpose: Use this subcommand to insert a comment line (such as a copyright notice) into your runfile, to be displayed by a LIST_EPF -EPF_DATA command. text-string can:

- Have a maximum length of 80 characters
- Use any characters, including valid separators

This subcommand may not be used on the PRIMOS command line.

▶ COMMON_WARNING

Abbreviation: CW

Purpose: Use this subcommand to check for the redefinition of COMMON blocks. This is the default mode; this subcommand reverses the effect of a NO_COMMON_WARNING subcommand. You receive a warning message if a common or external static variable is:

- Redefined to be a size smaller than a previous definition.
- Redefined to be a size larger than a previous definition before being allocated space in memory. (Typically, COMMON blocks are deferred, and hence are not allocated space in memory until a FILE or RESOLVE_DEFERRED_COMMON subcommand is issued.)

The default is for BIND to issue warning messages concerning the redefinition of the size of a common area.

▶ COMPRESS

Purpose: Use this subcommand to delete information not used in execution so that the runfile is compressed into a smaller file. The information deleted includes BIND's symbol table and DBG's information for debugging the program; as a result, the program may not be reloaded and maps may not be generated from existing compressed EPFs. Use this subcommand on programs that have already been debugged.

▶ DYNT list-of-names

Purpose: Use this subcommand to create dynamic links to your personal library routines or routines that may not be in the PFTNLB library.

To create a dynamic link to one or more library routines, list their names as arguments to the DYNT subcommand. Names that are already defined are ignored. The DYNT references are added to the EPF just as if they were in the object code linked via a LOAD or LIBRARY subcommand. If a name given is already defined in the BIND session, then a warning message is given only under the use of the set force load flag (SFL). Dynamic linking is discussed in detail in the Advanced Programmer's Guide, Volume I: BIND and EPFs.

▶ ENTRYNAME { -ALL | list-of-names | -NONE }

Abbreviation: EN

Purpose: Use the ENTRYNAME subcommand to define names as entrypoints into the library EPF runfile you are currently binding. You must have already issued the LIBMODE subcommand.

To specify entrypoints by name, use the LOAD subcommand to link the modules containing the entrypoints first, then use the ENTRYNAME list-of-names format to specify the entrypoints. Names not already linked are rejected with warning messages and are ignored.

To specify all subroutines in one or more modules as entrypoints, use the ENTRYNAME -ALL format. The -ALL option causes all subsequently linked subroutines to be named as entrypoints. After linking the desired modules, issue the ENTRYNAME -NONE subcommand so that subsequently linked subroutines are not made into entrypoints. In particular, issue the ENTRYNAME -NONE subcommand before issuing any LIBRARY subcommands.

For example:

```
LIBMODE -PROGRAM
LOAD SAMPLE1
LOAD SAMPLE2
LOAD SAMPLE3
ENTRYNAME SAMPLE1 SAMPLE2 SAMPLE3
LIBRARY
```

In the above example, the subroutines SAMPLE1, SAMPLE2, and SAMPLE3 are entrypoints to the library EPF. Other subroutines contained in the modules SAMPLE1, SAMPLE2, and SAMPLE3 are not made into entrypoints for the library EPF, and therefore are callable only by other entrypoints in the library EPF.

Another example:

```
LIBMODE -PROGRAM
ENTRYNAME -ALL
LOAD SAMPLE1
LOAD SAMPLE2
LOAD SAMPLE3
ENTRYNAME -NONE
LIBRARY
```

In this example, all of the subroutines in modules SAMPLE1, SAMPLE2, and SAMPLE3 are made entrypoints for the library EPF. The ENTRYNAME -NONE subcommand disables the automatic generation of entrypoints during the linking of the system libraries to prevent program errors.

▶ FILE [pathname]

Purpose: Use this subcommand to complete the processing of an EPF. The FILE subcommand automatically resolves any deferred common areas, writes the EPF into pathname.RUN, and then returns you to the PRIMOS command level.

BIND files the EPF in your directory with a .RUN suffix. If you specify pathname, BIND gives the runfile the name pathname.RUN. If you do not specify pathname, BIND uses either the name you specified on the BIND command line, or, if you specified no name, BIND uses the name of the first object module that you linked as the pathname and adds the suffix .RUN.

If you are using BIND on the command line, you do not have to issue this subcommand. By default, BIND appends the -FILE option to the end of the command line. Your runfile has either the name specified after the BIND command or the name of the first linked object module. (BIND adds the suffix .RUN.)

▶ HELP subcommand-name
-LIST

Purpose: Use this subcommand to get assistance while working with BIND. If you specify -LIST, BIND displays a list of all BIND subcommands and their abbreviations. For help with a particular subcommand, type HELP followed by the subcommand name.

▶ INITIALIZE_DATA [-OCTAL] initial_value

Abbreviation: IDATA

Purpose: Use this subcommand to initialize all uninitialized areas for debugging purposes. IDATA initializes data segments at the integer value of initial_value, a decimal value in the range -32768 to 32767. If -OCTAL (abbreviation: -OCT) is specified, the areas are initialized at an octal number from 0 to 177777. Initialization generates a larger runfile and increases startup time; therefore, it is recommended that this subcommand not be used for production copies.

▶ LIBMODE -PROGRAM
-PROCESS

Abbreviation: LM

Purpose: This subcommand specifies the mode, or class, of the library EPF being bound. If you specify -PROGRAM, the linkage area for the library is allocated the first time it is referenced by every program you invoke. Therefore, a program-class library EPF may have several copies of its linkage area allocated if several active programs are using it at a time for one user. If you specify -PROCESS, the library is given a linkage area only the first time a process references it. A process-class library EPF is initialized only one time during a login session, and only one copy of its linkage area exists at a time for one user. (See Chapter 6 for an explanation of process-class and program-class library EPFs.)

The following example shows the use of the LIBMODE subcommand to designate a program-class library:

```
LIBMODE -PROGRAM
```

After you designate the class of library, and before you issue the FILE subcommand, use the ENTRYNAME subcommand to create a table of entrypoints to the library.

▶ LIBRARY [link-spec-1 [link-spec-2] ...]

Abbreviation: LI

Purpose: Use this subcommand to link libraries. In most cases, link-spec-1, link-spec-2, and so on are simply the names of the libraries to be linked. However, link-spec-n has the format:

```
[link-option] library-name
```

See the description of link-option in the LOAD subcommand, described next. In most cases, you will not be specifying any link-option.

Each library-name in a link-spec-n may be either an entryname or a pathname. If library-name is an entryname, the LIBRARY subcommand looks for the library in the system-supplied directory called LIB, where all of Prime's standard libraries are kept. If library-name is a pathname, the LIBRARY subcommand looks for the library in the location you specify in the pathname.

If you do not supply any library-name, BIND links the standard system library PFTNLB to the EPF.

▶ LOAD link-spec-1 [link-spec-2] ...

Abbreviation: LO

Purpose: Use this subcommand to link the object files in object modules to the current EPF. If the EPF has not already been given a name, BIND uses the name you give in the first LOAD subcommand you issue. BIND attaches a .RUN suffix to the EPF in your directory when the processing of the runfile is finished.

The format of link-spec-n is:

```
[link-option] module-name
```

module-name is the pathname of an object module. If module-name is not a pathname, the current directory is scanned for the name. If you have two or more procedures in an object file, the procedures are linked in the order in which it they appear in the file.

link-option allows you to control the manner in which all subroutines in module-name are linked. Once a link-option is specified, it affects all subsequent module-names specified on the same BIND subcommand line.

Option	Meaning
{ -FORCE / -FO } pathname ...	Forcibly links all the procedure code in the files you list in module-name, thus overriding the RFL/SFL flags generated by the binary library editor, EDB.
{ -PAGE / -PA } pathname ...	Links all of the files you list in module-name beginning on page boundaries (procedure code only).
{ -FORCEPAGE / -FP } pathname ...	Forcibly links modules in the binary files you list in module-name on page boundaries (procedure code only).

If you do not use any options with the LOAD or LIBRARY subcommand, the default mode is for BIND to link all routines within a file until no unresolved references exist; this is the SFL (Set Force Load) flag. If the file contains an RFL (Reset Force Load) flag, only those routines previously referenced are loaded. (Regardless of the flag setting, the first routine in a file is always linked, and no padding is performed to place modules on page boundaries.)

Be aware that SFL will only forcibly load object text if there are some outstanding unresolved references. If all references have been resolved, nothing else is linked, whether you are in SFL or RFL mode. In this case, use the -FORCE option, above. Also, be aware that RFL will only load a module if it contains the definition of an unresolved reference. For more information about RFL and SFL, see the Advanced Programmer's Guide, Volume 1: BIND and EPFs.

Examples:

```
LOAD A                          /* links A normally (default)

LOAD A -FORCE B                 /* links A normally and
                                   force links B

LOAD -FORCE C -PAGE D E         /* force links C and links D
                                   and E on a page boundary

LOAD -FORCEPAGE F -FORCE G      /* force links F on a page
                                   boundary and force links G
```

▶ MAIN routine-name

Purpose: Use this subcommand to designate the main procedure of an EPF. Ordinarily, the main procedure of an EPF is the first routine that is linked. To designate some other routine as the main procedure, you can name it in a MAIN subcommand. However, a procedure must already be linked before it can be designated as the main procedure by this command. For example:

```
BIND MYPROG -LOAD SUBR1 SUBR2 MYPROG -MAIN MYPROG -LIBRARY
```

If you do not link the main procedure first or name it in a MAIN subcommand, execution of the EPF begins with whatever routine was linked first, and probably causes the program to fail. Use the MAP subcommand to see the main subroutine, displayed as START ECB on the first line of the map. Look up the subroutine by comparing the START ECB to the ECBs listed in the procedures section of the map.

With FTN, there is no such thing as a main procedure. There are only subroutines, which may be called MAIN.

With PMA, you must specify the main ECB of a module in the END pseudo-op. For example:

```
END MYPROG_ECB
```

▶ MAP [mapfile] [option]

Purpose: You use this subcommand to obtain a BIND map that gives you the following information on the EPF under construction:

- A description of the layout of your program in memory.
- The relative placement of procedure, linkage, and common areas with respect to one another in the EPF. (Segment numbers of linkage segments are negative numbers and those of procedure segments are positive. PRIMOS does not assign actual segment numbers until program runtime, and you may display these with the LIST_EPF -SEGMENTS command. The negative-numbered segments and positive-numbered segments in the LIST_EPF display correspond to the same segments in the BIND map, and correlate those segments with the actual segments assigned by PRIMOS.)

The arguments to the MAP subcommand are described below.

The Destination of the Map: You may specify one of the following values for the mapfile argument:

Value	Meaning
-TTY	The map is displayed at the terminal. This is the default mode if mapfile is not specified.
-SPOOL	The map is spooled under the name epf-name.MAP, where epf-name is the base name of the EPF under construction.
pathname	The map is written into the specified file. If the file already exists, it is overwritten.

Option: When no option is specified, the full map without the command processor flags is output. When an option is given, only the part of the map designated by the option is output. The possible options are:

Option	Meaning
-FULL	Standard map plus command processor flags
-FLAGS	Command processor flags currently set
-RANGES	Load ranges for each segment
-BASE	Base areas
{-UNDEFINED / -UN}	Symbols currently undefined
{-NAMED_SYMBOL / -NSY}	Named symbols sorted by name

Identifying Library EPF Entrypoints: MAP can also provide a list of the entrypoints that you have used when you are building a library EPF with BIND. MAP identifies these entrypoints by placing an asterisk (*) next to the entrypoint listed in the procedure area of the map. MAP also displays a message to this effect at the bottom of the map. For example:

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) 1988, Prime Computer, Inc.]
: LIBMODE -PROGRAM
Library is per program class.
: LO MAIN
: LO SUB
: ENTRYNAME SUB
: LI
BIND COMPLETE
: MAP
Map of MAIN                              (Map Version 1)

  START ECB: -0001/000000

  Segment Type              Low    High   Top
   -0002  DATA             000000 000073 000074
   +0000  PROC             001000 001263 001264

    Base Area: +0000 000100 000100 000777 000777

PROCEDURES:
  Name      ECB address  Initial PB% Stack size Link size Initial LB%
  .MAIN.   -0002/000010  +0000/001062 000062    000044    -0002/177400
*  SUB     -0002/000054  +0000/001222 000062    000030    -0002/177444

DYNAMIC LINKS:
  F$CB77    +0000/001240
  F$ILDR    +0000/001244
  F$ILDW    +0000/001250
  F$STOP    +0000/001254
  F$XFR     +0000/001260

COMMON AREAS:

OTHER SYMBOLS:

UNDEFINED SYMBOLS:

Note: * indicates a library entry point
: FILE MY_LIB
OK,
```

See the LIBMODE and ENTRYNAME subcommands earlier in this chapter for more information on entrypoints and library EPFs.

Producing a Retroactive Map: You are able to produce a map on an EPF that has already been built. This is useful, for example, if you want to debug an EPF but you do not have the map associated with it. For example:

```
OK, BIND
[BIND Rev. 22.0.B2 Copyright (c) 1988, Prime Computer, Inc.]
: LO MAIN.RUN
BIND COMPLETE
: MAP MAIN.MAP
: FILE
OK, SLIST MAIN.MAP
Map of MAIN                              (Map Version 1)

  START ECB: -0001/000000

  Segment Type              Low    High   Top
   -0002  DATA            000000 000073 000074
   +0000  PROC            001000 001263 001264

    Base Area: +0000 000100 000100 000777 000777

PROCEDURES:
  Name      ECB address   Initial PB% Stack size Link size Initial LB%
  .MAIN.   -0002/000010  +0000/001062 000062     000044    -0002/177400
*  SUB     -0002/000054  +0000/001222 000062     000030    -0002/177444

DYNAMIC LINKS:
  F$CB77     +0000/001240
  F$ILDR     +0000/001244
  F$ILDW     +0000/001250
  F$STOP     +0000/001254
  F$XFR      +0000/001260

COMMON AREAS:

OTHER SYMBOLS:

UNDEFINED SYMBOLS:

Note: * indicates a library entry point
OK,
```

Two restrictions in producing a map after the EPF has already been built are:

1) You cannot load an EPF after loading any code.

2) You cannot get a map of the EPF if you used the COMPRESS subcommand when the EPF was first produced, because COMPRESS does not copy the symbol table (which BIND maintains to produce maps) to the output file.

▶ NO_COMMON_WARNING

Abbreviation: NCW

Purpose: Use this subcommand to turn off the checking for redefinition of COMMON blocks. No warning messages are issued. An error message is issued only if a common area is redefined as larger after being allocated memory; the owning procedure is not linked.

▶ QUIT

Abbreviation: Q

Purpose: Use this subcommand to end your BIND session without saving or replacing the current EPF. If you have linked any code, BIND asks you to verify whether or not you want to quit without filing the EPF. Answer Y or YES if you are certain that you wish to leave BIND.

If you are using BIND on the command line with options, and no code has been linked, BIND appends a -QUIT option at the end of the command line for you. However, if you have linked code and you don't want to create an EPF, you must specify the -QUIT option. For example:

```
BIND -LOAD MYPROG.RUN -MAP MYPROG.MAP -QUIT
```

▶ RELOAD link-spec-1 [link-spec-2] ...

Abbreviation: RL

Purpose: Use this subcommand to relink a binary file into an existing EPF. You may use RL to replace a procedure in a larger program for testing and debugging without having to rebuild the entire program. The original copy of the procedure is not removed. The new version is added as a whole to the end of the existing procedure code, and appropriate pointers in the EPF are redirected.

The format of the RELOAD subcommand is the same as the LIBRARY and LOAD subcommands, including the use of the link-option. See the LOAD subcommand, above, for more information.

▶ RESOLVE_DEFERRED_COMMON

Abbreviation: RDC

Purpose: Use this subcommand to allocate space for common blocks that have been deferred. Using this subcommand resolves pointers to COMMON blocks. Use the RDC subcommand before a MAP subcommand to find the locations of the common blocks in your map.

▶ SEARCH_RULE_VERIFY { -ON | -OFF }

Abbreviation: SRVFY

Purpose: Use this subcommand to find out the full pathname of the object being loaded into the program with BIND. When you use BINARY$ search rules, it is sometimes difficult to tell from which directory in the file system a file is being loaded into your program. SEARCH_RULE_VERIFY displays the full pathname of any .RUN and/or .BIN object being loaded during the BIND operation; that is, SRFVY displays which .BIN and/or .RUN files are being loaded from which directories.

The default is SRVFY -OFF.

SRVFY Example: Following is an example of the SEARCH_RULE_VERIFY subcommand. First, set your search rules with the SET_SEARCH_RULES command to BINARY$.

```
OK, SSR BINARY$ -NS
```

Next, use the LIST_SEARCH_RULES command to check your search rules; this reveals the following pathnames under BINARY$:

```
List: BINARY$
Pathname of template: <SYSONE>TERRY.A>BINARY$.SR

    <sysone>terry.a
    <systwo>terry.b>programs
```

Finally, create an EPF with BIND using the SEARCH_RULE_VERIFY subcommand. Notice that BIND gives you the full pathnames of the files being loaded because SEARCH_RULE_VERIFY is enabled.

```
OK, BIND
[BIND Rev. 22.0 Copyright (c) 1987, Prime Computer, Inc.]
: SRVFY -ON
: LO SYMBOLS
Loading: <SYSONE>TERRY.A>SYMBOLS.BIN
: LI CBLLIB
Loading: <SYSONE>LIB>CBLLIB.BIN
: LI
Loading: <SYSONE>LIB>PFTNLB.BIN
BIND COMPLETE
: FILE
OK,
```

▶ SYMBOL symbol-name definition [size]

Abbreviation: SY

symbol-name	A name defined by the SYMBOL command
definition	The name of a symbol already defined or of an absolute address (segno/offset)
size	The size in halfwords of symbol_name

Purpose: Use this subcommand to:

- Equate the argument symbol-name to another symbol name that you define in definition. If symbol_name is a symbol already defined, BIND displays an error message. However, if symbol-name is an undefined symbol, it becomes a defined symbol after you use this subcommand. Using the SYMBOL subcommand in this way allows references to one procedure to be reinterpreted as references to some other procedure. (For example, SYMBOL '' segno/offset allows you to reference blank common as defined by F77.)

- Equate the argument symbol-name to an absolute address that you specify in definition as segno/offset. Any reference in your program to a data item in symbol-name is now a reference to the corresponding data item in segno/offset.

▶ VERSION character-string

Purpose: Use this subcommand to put a version indicator into your runfile. character-string can have a maximum of 32 characters, but may not contain valid separators unless the string is placed within single quotation marks. In order to use a single quotation mark as a valid separator, use two single quotation marks together. If you do not use single quotation marks, character string cannot include the following separators:

- Blanks
- Commas (,)
- Semicolons (;)
- Slash-asterisks (/*)

As the next example shows, the period and dash are valid characters:

```
VERSION 1.0-13
```

The following example illustrates how to include blanks and other valid separators:

```
VERSION 'Version 22, sub-unit 6'
```

EXECUTABLE PROGRAM FILES AS COMMANDS

This section describes the BIND subcommands available for creating EPFs as commands. EPFs have the ability either to process optional features (wildcards, name generation, or treewalking) within a command line or to let the command processor do it instead. An EPF command can also be built in such a way that it acts only on certain targets, such as files or access control lists.

Command line processing is done by calling the routines CL$PIX, CMDL$A, or RDTK$$. These subroutines are described in the Subroutines Reference series. For more information about command line features, see the PRIMOS Commands Reference Guide. For information on command line processing, see Chapter 10 of this manual and the Advanced Programmer's Guide, Volume III: Command Environment.

▶ ITERATION

Abbreviation: ITR

Purpose: Use this subcommand with BIND to create an EPF in which the command processor performs iteration when the runfile is invoked. This is the default.

▶ NAMGENPOS token-position

Abbreviation: NGP

Purpose: Use this subcommand with BIND to perform name generation on the argument in the token-position on the command line when invoking the runfile. The default is NAMGENPOS 1; the first token following the tokens that invoked the program is the name generation token. For example, in the command line RESUME MYPROG MYFILE.IN =.OUT, MYFILE.IN is token 1, and token 2 becomes MYFILE.OUT.

▶ NO_GENERATION

Abbreviation: NG

Purpose: Use this subcommand to tell the command processor not to perform name generation on the command line when invoking the runfile. Tokens containing = are passed to your program without modification.

▶ NO_ITERATION

Abbreviation: NITR

Purpose: Use this subcommand to create an EPF that tells the command processor to suppress iteration when the runfile is invoked. Parentheses on the command line are passed to your program without modification.

▶ NO_TREEWALK

Abbreviation: NTW

Purpose: Use this subcommand to create an EPF that tells the command processor not to perform treewalking when the runfile is invoked. If a pathname has wildcard elements in intermediate portions of the pathname, they are passed to your program without modification.

Note

If your program disables treewalking with the NO_TREEWALK subcommand, but does allow wildcarding, then a pathname with wildcard specifications in the final element of the pathname is expanded by the command processor. However, this expansion will not take place if an intermediate element of the pathname has a wildcard character. In this case, the entire pathname is passed to your program without modification.

▶ NO_WILDCARD

Abbreviation: NWC

Purpose: Use this subcommand to create an EPF that tells the command processor not to process wildcard expansion when the runfile is invoked. Instead, the final element of a pathname that contains the @, +, or characters is passed to your program without modification.

▶ TREEWALK

Abbreviation: TW

Purpose: Use this subcommand to instruct PRIMOS to perform treewalking when the runfile is invoked. This is the default mode.

▶ WILDCARD [selection-options] [verification-option]

Abbreviation: WC

Purpose: Use this subcommand to create an EPF in which the command processor expands wildcards when the EPF is invoked and to specify default file system object selection and verification options. You cannot use this subcommand on the PRIMOS command line.

Use the selection-options for the WILDCARD command to specify the types of objects to be selected for processing:

Option	Objects processed
-FILE	SAM, DAM, and CAM files
{ -DIRECTORY, -DIR }	Directories
{ -SEGMENT_DIRECTORY, -SEGDIR }	Segment directories
{ -ACCESS_CATEGORY, -ACAT }	Access categories
-RBF	ROAM Files

If you do not specify any of the options, the default mode is for the command to process all objects except for ROAM (-RBF) files.

You may also choose one of the following verify-options:

Option	Meaning
{ -VERIFY, -VFY }	PRIMOS asks for verification before each object matching a wildcard is processed.
{ -NO_VERIFY, -NVFY }	PRIMOS does not ask for verification before each object matching a wildcard is processed.

If you do not specify any of the options, the default mode is that verification is turned off (-NO_VERIFY).

The following example shows the use of the the -FILE and the -VERIFY options with the WILDCARD command:

```
WILDCARD -FILE -VERIFY
```

The WILDCARD subcommand selects the default options for a program, but a user invoking the program can override any of these options. If the user specifies at least one selection option during invocation, the user's selection completely overrides selection-options for that invocation. Similarly, if the user specifies the -VERIFY option, then verification of wildcard selections takes place even if you specified WILDCARD -NO_VERIFY during the BIND session.

9
EPF Commands Dictionary

OVERVIEW

This chapter is a dictionary of PRIMOS commands that relate to EPFs. In alphabetical order, these commands are:

- EXPAND_SEARCH_RULES
- INITIALIZE_COMMAND_ENVIRONMENT
- LIST_EPF
- LIST_LIBRARY_ENTRIES
- LIST_LIMITS
- LIST_MINI_COMMANDS
- LIST_SEARCH_RULES
- LIST_SEGMENT
- REMOVE_EPF
- SET_SEARCH_RULES

You can use the COPY command to replace an open EPF file with another EPF file, renaming the replaced file. This chapter also explains how to do this.

CHOOSING WHICH COMMAND TO USE

Table 9-1 provides a quick summary of the function of each of the commands described in this chapter, to help you select the right one for your purpose.

Table 9-1
The EPF Commands

Command	Use
COPY	Replace an open EPF with another EPF.
EXPAND_SEARCH_RULES	Get the full pathname of a file system object or search list.
INITIALIZE_COMMAND_ ENVIRONMENT	Reset your command environment to the state it is in when you first log in.
LIST_EPF	List information about any or all of the EPFs you are using.
LIST_LIBRARY_ENTRIES	List information on entrypoints in library EPFs.
LIST_LIMITS	Find out how many command levels, program invocations per command level, and private segments you can use.
LIST_MINI_COMMANDS	Find out what commands you can use at mini-command level.
LIST_SEARCH_RULES	List the contents of your entrypoint search list.
LIST_SEGMENT	Find out which segments you are using.
REMOVE_EPF	Remove an EPF from your address space.
SET_SEARCH_RULES	Specify an entrypoint search list. You need not do this unless you want to use a personal search list.

EPF MINI-COMMANDS

Your System or Project Administrator sets a limit on the number of command levels you can use. If you exceed this limit, you reach mini-command level. This is a command level from which you can use only a limited subset of PRIMOS commands, called the mini-commands. Table 9-2 lists these 17 commands.

When you reach mini-command level, PRIMOS displays the message:

```
You have exceeded your maximum number of command levels.

You are now at mini-command level.  Only the commands shown
below are available.  Of these, RLS -ALL should return you to
command level 1.  If it does not, type ICE.  If this problem
recurs, contact your System Administrator.

Valid mini-commands are:

Abbrev  Full name              Abbrev  Full name
------  ---------------------  ------  ------------------------------
C       CLOSE                  COMO    COMOUTPUT
DMSTK   DUMP_STACK             ICE     INITIALIZE_COMMAND_ENVIRONMENT
LE      LIST_EPF               LL      LIST_LIMITS
LMC     LIST_MINI_COMMANDS     LS      LIST_SEGMENT
        LOGIN                  LO      LOGOUT
P       PM                     PR      PRERR
        RDY                    REN     REENTER
RLS     RELEASE_LEVEL          REMEPF  REMOVE_EPF
S       START

OK,
```

Until you return to a command level within your limit, you can use only the mini-commands. In addition, your personal abbreviations are not enabled while you are at mini-command level. If you try to quit by typing CONTROL-P or another terminal quit character, the following message is displayed:

```
Terminal QUIT invalid now. (listen_)
```

This message is followed by the list of commands you can use at this level, shown in the previous example. Another way to display this list is to use the LIST_MINI_COMMANDS command. Table 9-2 also shows where each mini-command is documented. This chapter covers several of them; all the others are described in the PRIMOS Commands Reference Guide.

To find out more about mini-command level and ways to choose the best mini-command for your purposes, see Chapter 7, which explains some trouble-shooting techniques.

EPF-RELATED PRIMOS COMMANDS

This part of the chapter describes the ten EPF-specific PRIMOS commands, in alphabetical order. The description of each command starts on a new page.

Table 9-2
The Mini-commands

Command Abbreviation	Command Name	Described in:
C	CLOSE	PRIMOS Commands Reference Guide
COMO	COMOUTPUT	PRIMOS Commands Reference Guide
DMSTK	DUMP_STACK	PRIMOS Commands Reference Guide
ICE	INITIALIZE_COMMAND_ENVIRONMENT	This chapter
LE	LIST_EPF	This chapter
LMC	LIST_MINI_COMMANDS	This chapter
LL	LIST_LIMITS	This chapter
LS	LIST_SEGMENT	This chapter
LOGIN	LOGIN	PRIMOS Commands Reference Guide
LO	LOGOUT	PRIMOS Commands Reference Guide
P	PM	PRIMOS Commands Reference Guide
PR	PRERR	PRIMOS Commands Reference Guide
RDY	RDY	PRIMOS Commands Reference Guide
REMEPF	REMOVE_EPF	This chapter
REN	REENTER	PRIMOS Commands Reference Guide
RLS	RELEASE_LEVEL	PRIMOS Commands Reference Guide
S	START	PRIMOS Commands Reference Guide

▶ EXPAND_SEARCH_RULES [options]

Abbreviation: ESR

Options: object_name [-LIST_NAME listname]
-SUFFIX suffix
-FILE
-DIRECTORY
-SEGMENT_DIRECTORY
-ACCESS_CATEGORY
-REFERENCING_DIR pathname

Purpose: Use this command to provide the fully qualified pathname of a specified file system object or search list.

object_name is used to specify the objectname to be expanded, and object_name must include all suffixes. PRIMOS either returns the fully qualified pathname to the terminal screen, or issues a message indicating that the requested object could not be found.

Choosing Search Lists

EXPAND_SEARCH_RULES uses one of your search lists to determine the fully qualified pathname. Use the LIST_NAME option to specify the appropriate search list. If you do not specify a LIST_NAME option, EXPAND_SEARCH_RULES selects the appropriate search list based on the suffix of the objectname.

The following list shows how PRIMOS selects search lists by default:

Suffix	Search List
.RUN	COMMAND$
.SAVE	COMMAND$
.CPL	COMMAND$
Other/no suffix	ATTACH$

EXPAND_SEARCH_RULES Options

This section explains the options to EXPAND_SEARCH_RULES.

-ACCESS_CATEGORY
Specifies that the file system object sought is an access category. This option allows you to limit the search to access categories.

-DIRECTORY
Specifies that the file system object sought is a directory. This option allows you to limit the search to directories.

-FILE
Specifies that the file system object sought is a file. This option allows you to limit the search to files.

-LIST_NAME listname
Permits you to specify the name of the search list that PRIMOS should use to locate the named object. If you do not specify -LIST_NAME, PRIMOS selects the search list (ATTACH$ or COMMAND$) based on the suffix of the objectname. If you specify -LIST_NAME as COMMAND$, PRIMOS supplies suffixes to the objectname in the following sequence: .RUN, .SAVE, .CPL, no suffix.

-REFERENCING_DIR pathname
Permits you to specify a search rule that PRIMOS substitutes for the REFERENCING_DIR entries in the search list. EXPAND_SEARCH_RULES uses this search rule to search for the file system object.

-SEGMENT_DIRECTORY
Specifies that the file system object sought is a segment directory. This option allows you to limit the search to segment directories.

-SUFFIX suffix
Specifies suffixes that PRIMOS appends to the objectname to conduct the search. The suffixes must begin with a period (for example, .RUN). You can specify a maximum of eight suffixes following a -SUFFIX option. PRIMOS searches for suffixes in the following order: .RUN, .SAVE, .CPL. If no match is found with all listed suffixes, PRIMOS searches for the object with no suffix.

You may invoke EXPAND_SEARCH_RULES as a CPL function. In this case, EXPAND_SEARCH_RULES returns the fully qualified pathname to a variable in the CPL program. Also, you can invoke ESR by a subroutine call.

▶ INITIALIZE_COMMAND_ENVIRONMENT [-SERVER]

Abbreviation: ICE

Purpose: You use this command to reset your command environment to the state that it is in when you first log in. You may want to do this if you suspect that your command environment has been damaged in some way, or if you are having trouble getting out of mini-command level.

Before using this command, issue the command CLOSE -ALL followed by the command RELEASE_LEVEL -ALL. These two commands may solve the problem without reinitializing your environment.

When you use INITIALIZE_COMMAND_ENVIRONMENT, it performs the following functions:

1. De-allocates any remote NPX slave processes active for your user process.
2. Clears all X.25 virtual circuits open for you, except for the remote-login virtual circuit, if any.
3. Closes all your files, including a COMOUTPUT file if you have one open.
4. Frees all your private dynamic and static segments.
5. Resets the Ring 3 command environment to an initial state.
6. Reattaches you to your origin directory (initial attach point).
7. Executes your login file (LOGIN.RUN, LOGIN.SAVE, LOGIN.COMI or LOGIN.CPL) in your origin directory. Your login file usually sets up your abbreviations, defines your erase and kill characters, sets global variables, and so on.

The -SERVER Option: When you use the -SERVER option, ICE performs the seven operations listed above, and also performs the following:

1. Terminates all of the server's InterServer Communication (ISC) sessions.
2. Terminates all PRIMIX child processes that are a part of the server. This server is the server to which the caller's process belongs.
3. Deletes all synchronizers and timers.

-SERVER is available only to a terminal process or phantom process; it is not available to a child process.

Subsystem Notification: INITIALIZE_COMMAND_ENVIRONMENT does not notify other programs or subsystems that it is about to re-initialize your environment. That is, INITIALIZE_COMMAND_ENVIRONMENT terminates programs that expect the CLEANUP$ condition, without signaling the condition, so that these programs have no opportunity to clean up.

The reason for this is that if a user's PRIMOS command environment has been damaged, that user's command stack may also have been damaged. If this has happened, signaling a condition such as CLEANUP$ to give other procedures a chance to perform last-minute abort processing before the command environment is re-initialized is not likely to work. This means that INITIALIZE_COMMAND_ENVIRONMENT, like the LOGOUT command, does not signal the condition CLEANUP$ before unwinding the command stack.

In the following example, a user reaches mini-command level, and PRIMOS automatically lists all the mini-commands. The user then gives the STATUS command, which fails. Finally, the user initializes the command environment.

```
OK,
QUIT.
You have exceeded your maximum number of command levels.

You are now at mini-command level.  Only the commands shown
below are available.  Of these, RLS -ALL should return you to
command level 1.  If it does not, type ICE.  If this problem
recurs, contact your System Administrator.

Valid mini-commands are:

Abbrev  Full name               Abbrev  Full name
------  ---------------------   ------  ------------------------------
C       CLOSE                   COMO    COMOUTPUT
DMSTK   DUMP_STACK              ICE     INITIALIZE_COMMAND_ENVIRONMENT
LE      LIST_EPF                LL      LIST_LIMITS
LMC     LIST_MINI_COMMANDS      LS      LIST_SEGMENT
        LOGIN                   LO      LOGOUT
P       PM                      PR      PRERR
        RDY                     REN     REENTER
RLS     RELEASE_LEVEL           REMEPF  REMOVE_EPF
S       START

OK, STATUS
Invalid command "STATUS" - only mini level commands accepted. (min$cp)
ER! INITIALIZE_COMMAND_ENVIRONMENT
OK,
```

▶ LIST_EPF [pathname-1 [... pathname-8]] [options]

Abbreviation: LE

Options:		
	-ACTIVE	-AC
	-COMMAND_PROCESSING	-CP
	-DETAIL	-DET
	-EPF_DATA	-ED
	-HELP	-H
	-LIBRARY	-LI
	-NOT_ACTIVE	-NA
	-NOT_MAPPED	-NM
	-NO_WAIT	-NW
	-PROGRAM	-PRG
	-SEGMENTS	-SEGS

Purpose: You use this command to display information on EPFs. The LIST_EPF command can display information about EPFs whether or not the EPF is currently mapped to your address space. That is, the command works on two domains:

- Your address space. You can display information on any or all of the EPFs that are mapped into your address space.
- The file system. You can display information on any EPF by giving its pathname. If you want to find out about an EPF that is not mapped to your address space, use the -NOT_MAPPED option, described below.

Unless you use the -NOT_MAPPED option, LIST_EPF looks for an EPF that is already mapped into your address space. If you give the pathname of an EPF that is not mapped and do not specify the -NOT_MAPPED option, LIST_EPF displays no information about that EPF.

You can use pathname-1 through pathname-8 to specify a maximum of eight pathnames of EPFs. You need not include the EPF suffixes .RUN or .RPn, (where n is a digit ranging 0 through 9.) These pathnames may be simple filenames or full pathnames.

You may include wildcards within the final component of the pathname, but LIST_EPF does not support treewalking or iteration.

This example shows how to use LIST_EPF with a pathname:

```
OK, LIST_EPF <MKT>LIBRARIES*>@@

1 Process-Class Library EPF.

(active)     <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN

1 Program-Class Library EPF.

(active)     <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN

OK,
```

If you give the command with no options, LIST_EPF displays the full pathname of the EPF file or files, and sorts the files by type. This type may be one of:

- Program EPF
- Program-class Library EPF
- Process-class Library EPF

Within these types, LIST_EPF displays the names in alphabetical order based on the filename of the EPF. The names of the EPFs displayed include the .RUN or any of the .RPn suffixes, whichever applies.

LIST_EPF also displays the status of each EPF. The status of an EPF may be one of these three: active, not active, or not mapped.

Active: PRIMOS treats an EPF as active if it is either executing or suspended. This could mean one of the following:

- The EPF is a program or program-class library EPF that has been suspended while executing.
- The EPF is a program-class library EPF that has been invoked by a suspended program. Even though the library EPF itself is not suspended, it is considered active as long as its invoking program is active.
- The EPF is a process-class library, and has been initialized. This is also called an in-use EPF.

Not active: PRIMOS treats an EPF as not active if it has completed its execution and is still mapped into the user's address space.

Not mapped: PRIMOS classifies all other EPFs as not mapped.

If you use LIST_EPF with a filename rather than a pathname, the command displays the full pathnames of all the EPFs with that filename, as this example shows:

```
OK, LIST_EPF LD

2 Program EPFs.

(not active) <MKT>CMDNC0>LD.RUN
(not active) <MK2>LIZ>LD.RUN

OK,
```

If you do not specify a pathname, LIST_EPF displays information for all EPFs currently mapped to your address space, as shown in the following example:

```
OK, LIST_EPF

1 Process-Class Library EPF.

(active)     <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN

3 Program EPFs.

(active)     <MKT>CMDNC0>LD.RUN
(active)     <MK2>LIZ>LD.RUN
(active)     <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN

OK,
```

If the file you specify in pathname does not exist, you see the message shown in the following example:

```
OK, LIST_EPF SPENSER>FAERIE_QUEEN

No entries selected.

OK,
```

Options That Select the Type of EPF

To select the type or status of EPF to be listed, you can use the following options:

Option	Meaning
{ -ACTIVE, -AC }	Selects only active EPFs.
{ -NOT_ACTIVE, -NA }	Selects only non-active EPFs.
{ -NOT_MAPPED, -NM }	Displays information on the EPF specified by *pathname*. If no pathname is specified, displays information on all EPF files in the user's current (working) directory.
{ -PROGRAM, -PRG }	Selects only program EPFs.
{ -LIBRARY, -LI }	Selects only library EPFs.

The following examples show the use of the -ACTIVE and -NOT_ACTIVE options.

```
OK, LIST_EPF -ACTIVE

1 Process-Class Library EPF.

(active)      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN

3 Program EPFs.

(active)      <MKT>CMDNC0>LD.RUN
(active)      <MK2>LIZ>LD.RUN
(active)      <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN

OK, LIST_EPF -NOT_ACTIVE

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN

OK,
```

The -SEGMENTS Option

To find out what segments and linkage areas your EPFs are using, specify the -SEGMENTS option, abbreviated -SEGS. For all EPFs that are currently mapped into your address space, LIST_EPF -SEGMENTS displays:

1. The type of the EPF.
2. The status of the EPF.
3. The full pathname of the EPF.
4. The number of procedure segments being used by the EPF.
5. For each procedure segment in use, two numbers separated by a colon. The number to the left of the colon is an even integer greater than or equal to zero, preceded by a plus (+) sign. The number to the right of the colon shows the actual segment number used for the EPF procedure. The integer on the left relates the actual segment number to the imaginary segment number indicated by the same even integer in the BIND map for the EPF.
6. The number of linkage areas being used by the EPF.
7. For each linkage area in use, two numbers separated by a colon. The number to the left of the colon is an even integer less than zero, preceded by a minus (-) sign. The number to the right of the colon shows the segment/offset pair of the linkage area used for the most recent invocation of the EPF. The integer on the left relates the actual segment number to the imaginary segment number indicated by the same integer in the BIND map for the EPF.

If procedure segments or linkage areas have not yet been allocated to the EPF, the phrase "not allocated" is displayed.

The following examples show the use of the -SEGMENTS option, both with and without a filename.

```
OK, LIST_EPF -SEGMENTS

1 Process-Class Library EPF.

(active)      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
  2 procedure segments:    +0:4152             +2:4153
  2 linkage areas:         -2:4376(0)/0        -4:4377(3)/674

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
  1 procedure segment:     +0:4154
  1 linkage area:           (not allocated)

3 Program EPFs.

(active)      <MKT>CMDNC0>LD.RUN
  1 procedure segment:     +0:4156
  1 linkage area:          -2:4377(3)/113474
(active)      <MK2>LIZ>LD.RUN
  1 procedure segment:     +0:4157
--More--(CR)
  1 linkage area:          -2:4377(3)/34742
(active)      <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN
  1 procedure segment:     +0:4155
  1 linkage area:          -2:4377(3)/34066

OK, LIST_EPF MYPROG -SEGMENTS

1 Program EPF.

(active)      <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN
  1 procedure segment:     +0:4155
  1 linkage area:          -2:4377(3)/34066

OK,
```

The following example shows the use of the -SEGMENTS and the -NOT_MAPPED options together, for a user attached to the directory CMDNC0.

```
OK, LIST_EPF COPY -NOT_MAPPED -SEGMENTS

1 Program EPF.

(not mapped) <MKT>CMDNC0>COPY.RUN
  1 procedure segment:     (not allocated)
  1 linkage area:          (not allocated)

OK,
```

The -COMMAND_PROCESSING Option

To look at the state of command processing features for a program EPF, use the -COMMAND_PROCESSING option, abbreviated -CP. The LIST_EPF -COMMAND_PROCESSING command displays the full pathname of the EPF and, for a program EPF, displays command processing features, such as:

- The type of file system objects on which the EPF operates
- Whether the command processor should process wildcarding, treewalking, or command iteration for the EPF concerned
- The name generation position for the EPF

For the first two categories, the presence of the terms indicates that the feature is enabled. Command processing information is not relevant for library EPF, and is not displayed. For example:

```
OK, LIST_EPF -COMMAND_PROCESSING

1 Process-Class Library EPF.

(active)     <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN

3 Program EPFs.

(active)     <MKT>CMDNC0>LD.RUN
  command options:  trwlk,iter        file,dir,segdir,acat      1
(active)     <MK2>LIZ>LD.RUN
  command options:  trwlk,iter        file,dir,segdir,acat      1
(active)     <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN
  command options:  (none)            (none)                    1

OK,
```

The -EPF_DATA Option

To display general information on an EPF, use the -EPF_DATA option, abbreviated -ED. The LIST_EPF -EPF_DATA command displays the following information for the specified EPF or EPFs:

1. The type, status, and full pathname of the EPF
2. The version of BIND used to create the EPF
3. The date on which the EPF was bound
4. The program name of the EPF
5. The user version of the EPF
6. The contents of the EPF comment field
7. The number of debugger segments being used by the EPF

For example:

```
OK, LIST_EPF -EPF_DATA

1 Process-Class Library EPF.

(active)      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
  bind version:      19.4.0AV
  date of binding:   84-10-17.15:38:28.Wed
  program name:      SYSTEM_LIBRARY
  user version:      (none)
  comment:           Copyright (C) 1983, Prime Computer, Inc.,
                     Natick, Ma. 01760 All rights reserved
  debug segments:    2

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
  bind version:      19.4.0AV
  date of binding:   84-10-17.15:40:52.Wed
  program name:      FORTRAN_IO_LIBRARY
  user version:      19.4.10TR
  comment:           Copyright (C) 1983, Prime Computer, Inc.,
                     Natick, Ma. 01760 All rights reserved
  debug segments:    1
--More--(CR)

1 Program EPF.

(active)      <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN
  bind version:      19.4.0AE
  date of binding:   84-05-31.14:43:40.Thu
  program name:      POWERS
  user version:      (none)
  comment:           (none)
  debug segments:    1

OK,
```

If the EPF was bound by a version of BIND that cannot supply data on items 2 through 6, this message is displayed instead:

```
EPF data not available.
```

(Some PRIMOS commands were created as EPFs prior to Rev. 19.4; the version of BIND used to create them may not supply this information.)

The -DETAIL Option

To display all information on an EPF, use the -DETAIL option, abbreviated -DET. This option displays all attributes for each entry selected. These attributes include those displayed by the -COMMAND_PROCESSING, -EPF_DATA and -SEGMENTS options. For example:

```
OK, LIST_EPF -DETAIL

1 Process-Class Library EPF.

(active)      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
  2 procedure segments:    +0:4152              +2:4153
  2 linkage areas:         -2:4376(0)/0         -4:4377(3)/674
  bind version:        19.4.0AV
  date of binding:     84-10-17.15:38:28.Wed
  program name:        SYSTEM_LIBRARY
  user version:        (none)
  comment:             Copyright (C) 1983, Prime Computer, Inc.,
                       Natick, Ma. 01760 All rights reserved
  debug segments:      2


1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
  1 procedure segment:    +0:4154
  1 linkage area:           (not allocated)
--More--(CR)
  bind version:        19.4.0AV
  date of binding:     84-10-17.15:40:52.Wed
  program name:        FORTRAN_IO_LIBRARY
  user version:        19.4.10TR
  comment:             Copyright (C) 1983, Prime Computer, Inc.,
                       Natick, Ma. 01760 All rights reserved
  debug segments:      1


1 Program EPF.

(active)      <MK2>LIZ>TIM>SAMUEL>SUB1>SUB2>SUB3>MYPROG.RUN
  1 procedure segment:    +0:4155
  1 linkage area:         -2:4377(3)/34066
  bind version:        19.4.0AE
  date of binding:     84-05-31.14:43:40.Thu
  program name:        POWERS
  user version:        (none)
  comment :            (none)
  debug segments:      1
  command options:     wldcrd,trwlk,iter   file,dir,segdir,acat     1


OK,
```

The -NO_WAIT and -HELP Options

To control screen scrolling, use the -NO_WAIT option, abbreviated -NW. To remind yourself of the syntax of the command, use the -HELP option, abbreviated -H.

The -NO_WAIT option enables terminal screen scrolling. This option suppresses the --More-- prompt that is otherwise given at the end of each 23 lines of display.

If you do not specify -NO_WAIT, PRIMOS prompts you before scrolling the terminal screen. To display the next screen of output, press RETURN or enter Y, YES, OK, or NEXT. To exit from the command, enter N, NO, Q, or QUIT.

The -HELP option displays the syntax of LIST_EPF. This display is also printed if PRIMOS encounters an error while parsing the command.

▶ LIST_LIBRARY_ENTRIES [pathname-1 [... pathname-8]] [options]

Abbreviation: LLENT

Options:	-ENTRYNAME	-EN
	-HELP	-H
	-NO_WAIT	-NW

Purpose: You use this command to display alphabetically sorted selected entrypoints in a library EPF, where pathname identifies the library EPF. You can use the optional entrynames to select the library entrypoints to display. The entrynames can include wildcards.

You can use pathname-1 through pathname-8 to specify a maximum of eight pathnames of library EPFs. You may specify wildcard names. (Treewalking, however, is not supported.) You do not have to include the suffixes .RUN or .RPn.

If you do not specify any pathnames, the command displays information on all library EPFs listed in your entrypoint search list (ENTRY$.SR). If a library EPF specified in your entrypoint search list cannot be located, a warning message is displayed.

The default LIST_LIBRARY_ENTRIES output for each EPF displays a two-line header followed by an alphabetical listing of the entrypoints. The first line of the header lists the EPF's status (active, not active, or not mapped) and its pathname. The second line of the header lists the EPF's type (process-class or program-class), the total number of entrypoints, and the number of entrypoints currently selected for display.

If there are two or more EPFs, they are listed alphabetically.

By default, LIST_LIBRARY_ENTRIES displays entrypoints in seven columns per line of display. If the name of an entrypoint runs into an adjacent column, fewer than seven names are displayed on the affected line. For example:

```
OK, LIST_LIBRARY_ENTRIES LIBRARIES*>FORTRAN_IO_LIBRARY -EN I@@

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
Program-Class Library EPF,  115 Total Entrypoints, 16 Selected Entrypoints

I$AA01      I$AA12      I$AD07      I$AM05      I$AM10      I$AM11      I$AM13
I$AP02      I$BD07      I$BD7X      I$BM05      I$BM10      IOC$RA      IOCS$T
IOCS$_FREE_LOGICAL_UNIT             IOCS$_GET_LOGICAL_UNIT

OK,
```

The -ENTRYNAME Option

If you want to display information only on selected entrypoints within an EPF library, use the -ENTRYNAME option, abbreviated -EN:

```
{ -ENTRYNAME }  entryname-1 [entryname-2 ... entryname-8]
{ -EN        }
```

The -ENTRYNAME option limits the display to selected entrypoints within a library EPF. You use the entrynames, which may include wildcards, to specify which entrypoints you want displayed. If you do not specify entrynames or if you do not use the -ENTRYNAME option, all of the entrypoints within the library EPF are displayed.

The -NO_WAIT and -HELP Options

To control screen scrolling, use the -NO_WAIT option, abbreviated -NW. To remind yourself of the syntax of the command, use the -HELP option, abbreviated -H.

The -NO_WAIT option enables terminal screen scrolling. This option suppresses the --More-- prompt that is otherwise given at the end of each 23 lines of display.

If you do not specify -NO_WAIT, PRIMOS prompts you before scrolling the terminal screen. To display the next screen of output, press RETURN or enter Y, YES, OK, or NEXT. To exit from the command, enter N, NO, Q, or QUIT.

The -HELP option displays the syntax of LIST_LIBRARY_ENTRIES. This display is also printed if PRIMOS encounters an error while parsing the command.

▶ LIST_LIMITS

Abbreviation: LL

Purpose: You use this command to display information on various attributes affecting your command environment. The LIST_LIMITS command displays the following attributes:

- The number of command levels you can use
- The number of programs you can invoke at any command level
- The number of private dynamic segments you can use
- The number of private static segments you can use

LIST_LIMITS is useful when you think you may have exceeded one of these limits. For example, if you have used up all the command levels allocated to you and have reached mini-command level, you can use this command to check your limit.

Your System or Project Administrator sets all these attributes either in your user profile or on a system-wide basis.

The following example shows how LIST_LIMITS displays this information.

```
OK, LIST_LIMITS

Maximum number of command levels: 10
Maximum number of program invocations: 10
Maximum number of private static segments: 100
Maximum number of private dynamic segments: 150

OK,
```

Chapter 7, which discusses some troubleshooting techniques, explains the attributes of the command environment in more detail.

▶ LIST_MINI_COMMANDS [command-match]

Abbreviation: LMC

Purpose: Use this command to display the names of the PRIMOS commands that you can use after you have reached mini-command level, explained earlier in this chapter.

command-match is a character string that is used as a pattern match for mini-commands to be listed. The character string can contain wildcard characters. If you do not specify command-match, LIST_MINI_COMMANDS displays the names of all the PRIMOS commands that you can use at mini-command level.

For example, if you do not specify a command match:

```
OK, LIST_MINI_COMMANDS

Abbrev  Full name              Abbrev  Full name
------  ---------------------  ------  ------------------------------
C       CLOSE                  COMO    COMOUTPUT
DMSTK   DUMP_STACK             ICE     INITIALIZE_COMMAND_ENVIRONMENT
LE      LIST_EPF               LL      LIST_LIMITS
LMC     LIST_MINI_COMMANDS     LS      LIST_SEGMENT
        LOGIN                  LO      LOGOUT
P       PM                     PR      PRERR
        RDY                    REN     REENTER
RLS     RELEASE_LEVEL          REMEPF  REMOVE_EPF
S       START

OK,
```

If you specify the command match LIST@@:

```
OK, LIST_MINI_COMMANDS LIST@@
Abbrev   Full name                          Abbrev   Full name
-----------------------------------------------------------------

LE       LIST_EPF                           LL       LIST_LIMITS
LMC      LIST_MINI_COMMANDS                 LS       LIST_SEGMENT

OK,
```

```
▶ LIST_SEARCH_RULES   listname1...listname16
                      -NO_WAIT
                      -HELP
```

Abbreviation: LSR

Purpose: Use this command to display the contents of your active search lists, and to show the pathname of the search rules file used to create each search list.

LIST_SEARCH_RULES Options

The listname argument allows you to specify which search lists you wish to see. If you specify one or more list names, LIST_SEARCH_RULES displays those search lists in the order specified. You can specify a maximum of 16 list names. If you do not specify listname, LIST_SEARCH_RULES displays the five system-defined search lists mentioned below, plus all of the search lists that you have defined. The most recently set search list is displayed first.

The -NO_WAIT option causes terminal output to scroll continuously.

The -HELP option displays the command syntax and options. A help screen is also displayed if PRIMOS detects an error while parsing the command.

Default Search Lists

LSR lists the five system-defined search rule files by default. These five search lists are:

Search List	Purpose
ATTACH$	Searches partitions for top-level directories.
COMMAND$	Searches directories for executable code files.
ENTRY$	Searches library EPF files for entrypoints.
INCLUDE$	Searches directories for source code files.
BINARY$	Searches directories for binary object code files.

PRIMOS (by default) establishes the above special-purpose search lists when you either log in or otherwise initialize a process.

The following example illustrates the LIST_SEARCH_RULES command:

```
OK, LSR

List: ATTACH$
Pathname of template: <DISK2>SEARCH_RULES*>ATTACH$.SR

    -added_disks

List: INCLUDE$
Pathname of template: <DISK2>SEARCH_RULES*>INCLUDE$.SR

    [HOME_DIR]

List: BINARY$
Pathname of template: <DISK2>SEARCH_RULES*>BINARY$.SR

    [HOME_DIR]

List: COMMAND$
Pathname of template: <DISK2>SEARCH_RULES*>COMMAND$.SR

    cmdnc0

List: ENTRY$
Pathname of template: <DISK2>SEARCH_RULES*>ENTRY$.SR

    -primos_direct_entries
    libraries*>system_library.run
    libraries*>fortran_io_library.run
    libraries*>application_library.run
    -static_mode_libraries
OK,
```

▶ LIST_SEGMENT [segno-1 [... segno-8]] [options]

Abbreviation: LS

Options:		
	-BRIEF	-BR
	-DYNAMIC	-DY
	-HELP	-H
	-NAME	
	-NO_WAIT	-NW
	-STATIC	-ST

Purpose: The LIST_SEGMENT command displays information about the current user's private segments that are in use. The command displays only the private segments in the range '4000 through '5777.

Segment numbers are displayed in ascending numerical order. If you give the command with no options, then LIST_SEGMENT displays only the segment number and the access rights assigned to each segment.

Two possible combinations of access rights may be displayed:

Access Code	Access Allowed
RX	Read and execute access
RWX	Read, write, and execute access

For example:

```
OK, LIST_SEGMENT

1 Private static segment.
segment access
--------------
4000    RWX

8 Private dynamic segments.
segment access
--------------
4152    RX
4153    RX
4154    RX
4155    RX
4156    RX
4157    RX
4376    RWX
4377    RWX

OK,
```

You use segno-1 through segno-8 to specify a maximum of eight octal segment numbers on which you want information. You cannot include wildcards in the segment numbers or use iteration with them. If you do not give any segment numbers, LIST_SEGMENT displays information for each segment that is currently in use in the static and dynamic segment ranges.

The following examples show two uses of LIST_SEGMENT with specified segment numbers.

```
OK, LIST_SEGMENT 4152 4153 4000

1 Private static segment.
segment access
--------------
4000     RWX

2 Private dynamic segments.
segment access
--------------
4152     RX
4153     RX

OK, LIST_SEGMENT 4157 4160 4173

2 Private dynamic segments.
segment access
--------------
4157     RX
4160     RX

Private dynamic segment 4173 is not currently in use.

OK,
```

LIST_SEGMENT Options

Following are the options of the LIST_SEGMENT command:

Option	Meaning
-BRIEF -BR	Displays only the total number of segments that are currently in use in each segment range.
-DYNAMIC -DY	Displays information only about dynamic private segments.

Option	Meaning
-HELP -H	Displays the syntax of the LIST_SEGMENT command. The help display is also shown if PRIMOS encounters an error while parsing the command.
-NAME	Displays the name of any EPF file that is associated with the segment. (An EPF may be associated with a segment if the procedure or the linkage areas for that EPF are assigned to that segment.) This option is valid only for your own private, dynamic segments. If more than one EPF is associated with a given segment, as may happen if the linkage areas of several EPFs are allocated within the same segment, then the EPF pathnames are displayed alphabetically by filename on separate lines. If a given EPF uses more than one segment, the EPF pathname appears alongside the segment number/access right-hand pair for each segment. If a dynamic segment is not associated with an EPF, the word "none" appears by that segment.
-NO_WAIT -NW	Enables terminal screen scrolling. This option suppresses the --More-- prompt that is otherwise given at the end of each 23 lines of display. If you do not specify -NO_WAIT, PRIMOS prompts you before scrolling the terminal screen. To display the next screen of output, press RETURN or enter Y, YES, OK, NEXT. To exit from the command, enter N, NO, Q, or QUIT.
-STATIC -ST	Displays information only about static private segments.

LIST_SEGMENT Examples

The following example shows the use of the -NAME option.

```
OK, LIST_SEGMENT -NAME

1 Private static segment.
segment access
--------------
4000    RWX

11 Private dynamic segments.
segment access  epf
-------------------
4152    RX      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
4153    RX      <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
4154    RX      <MK2>LIZ>POWERS.RUN
4155    RX      <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
4156    RX      <MKT>CMDNC0>LD.RUN
4157    RX      <MKT>CMDNC0>COPY.RUN
4160    RX      <MKT>CMDNC0>DELETE.RUN
4161    RX      <MKT>CMDNC0>HINT.RUN
4375    RWX     <MKT>CMDNC0>DELETE.RUN
4376    RWX     <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
4377    RWX     <MKT>CMDNC0>COPY.RUN
                <MKT>CMDNC0>DELETE.RUN
                <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
                <MK2>LIZ>POWERS.RUN
                <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN
--More--(CR)
                <MKT>CMDNC0>HINT.RUN

OK,
```

The next example shows the summary display provided by the -BRIEF option.

```
OK, LIST_SEGMENT -BRIEF

1 Private static segment.

11 Private dynamic segments.

OK,
```

▶ REMOVE_EPF [pathname] [options]

Abbreviation: REMEPF

Options:	-ACTIVE	-AC
	-HELP	-H
	-NOT_ACTIVE	-NA
	-NO_QUERY	-NQ
	-NO_VERIFY	-NVFY
	-QUERY	-Q
	-VERIFY	-VFY

Purpose: You use this command to remove an EPF from your address space. That is, if an EPF is mapped to your address space, REMOVE_EPF unmaps it. REMOVE_EPF does not remove suspended EPFs.

REMOVE_EPF does not delete the EPF file itself. The command is useful at the following times:

- You want to allow another user to delete an EPF that you currently have mapped to your address space. Before the other user can use the DELETE command, you have to remove the EPF. If the other user tries to delete the EPF without first removing it, the other user receives an error message.
- You want to deallocate segments associated with one or more EPFs.

You can specify the pathname of the EPF file you want to remove. You can use a simple filename or a full pathname. You do not have to include the .RUN or .RPn suffixes. Because REMOVE_EPF supports command processor iteration, the pathname can include wildcards. The command does not, however, support treewalking.

If you do not give a pathname, REMOVE_EPF assumes that you want to remove all the non-suspended EPFs in your address space. The command asks you which EPFs you wish to remove, as shown in the following example. To remove the EPF, answer Y or YES. To leave the EPF alone, answer N or NO.

```
OK, REMOVE_EPF
Ok to remove EPF file <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN? NO
Ok to remove EPF file <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN? NO
Ok to remove EPF file <MKT>CMDNC0>DELETE.RUN? NO
Ok to remove EPF file <MKT>CMDNC0>LD.RUN? NO
Ok to remove EPF file <MKT>CMDNC0>HINT.RUN? NO

No EPFs removed (REMOVE_EPF).
OK,
```

If the EPF file you specify either does not exist or is not mapped into your address space, you receive the message shown in this example:

```
OK, REMOVE_EPF MISTAKE

No EPFs removed (REMOVE_EPF).
OK,
```

The options for REMOVE_EPF are:

Option	Meaning
{ -ACTIVE / -AC }	Terminates only active process-class library EPFs that are not in use by a suspended program or library. You cannot use REMOVE_EPF to remove suspended EPFs.
{ -NOT_ACTIVE / -NA }	Terminates only non-active EPFs. These are EPFs that are currently mapped to the user's address space but that are neither suspended EPFs nor in-use process-class library EPFs.
{ -VERIFY / -VFY }	Requests the user to verify all EPF terminations. By default, you are asked to verify terminations only when you include wildcards in pathname.
{ -NO_VERIFY / -NVFY }	Suppresses verification checking when you include wildcards in the pathname. You cannot use the -VERIFY and -NO_VERIFY options together.
{ -QUERY / -Q }	Requests the user to verify that an EPF is to be removed if the EPF is currently in use within the user's address space. This is the default.
{ -NO_QUERY / -NQ }	Suppresses user verification if the EPF is currently in use within the user's address space. You cannot use the -QUERY and -NO_QUERY options together.
{ -HELP / -H }	Displays the syntax of REMOVE_EPF. The help display is also printed if PRIMOS encounters an error while parsing the command.

In the following example, the user lists her EPFs, removes a FORTRAN library EPF, and finally removes all her inactive EPFs.

```
OK, LIST_EPF

1 Process-Class Library EPF.

(active)       <MKT>LIBRARIES*>SYSTEM_LIBRARY.RUN

1 Program-Class Library EPF.

(not active) <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN

5 Program EPFs.

(active)       <MKT>CMDNCO>COPY.RUN
(not active) <MKT>CMDNCO>DELETE.RUN
(not active) <MKT>CMDNCO>LD.RUN
(active)       <MK2>LIZ>POWERS.RUN
(not active) <MKT>CMDNCO>HINT.RUN

OK, REMOVE_EPF F@@
Ok to remove EPF file <MKT>LIBRARIES*>FORTRAN_IO_LIBRARY.RUN? YES
OK, REMOVE_EPF -NOT_ACTIVE
Ok to remove EPF file <MKT>CMDNCO>DELETE.RUN? YES
Ok to remove EPF file <MKT>CMDNCO>LD.RUN? YES
Ok to remove EPF file <MKT>CMDNCO>HINT.RUN? YES
OK,
```

▶ SET_SEARCH_RULES { -DEFAULT [-LIST_NAME listname] | pathname [-LIST_NAME listname] [-NO_SYSTEM] | -HELP }

Abbreviation: SSR

Options:		
	-DEFAULT	-DFLT
	-HELP	-H
	-LIST_NAME	-LNAM
	-NO_SYSTEM	-NS

Purpose: Use SET_SEARCH_RULES to copy a desired set of search rules from a search rules file into a search list.

A search rule is a location in the file system. (This location typically points to an object that you will likely be using; this object can be, for example, an EPF or a top-level directory.) A search rules file contains a sequential list of these search rules. In order to use this sequential list, you must copy it into memory. The sequential list in memory is called a search list. The action of copying the list of search rules into memory is called setting the search list, and this is done with the SET_SEARCH_RULES command.

The options for SET_SEARCH_RULES are:

Option	Meaning
{-DEFAULT | -DFLT} listname	Replaces the existing search list with a list of your own choosing. This option sets your search rules to a private list or resets your search rules to the system defaults.
{-HELP | -H}	Displays the syntax of the command. The help display is also shown if PRIMOS encounters an error while parsing the command.
{-LIST_NAME | -LNAM} listname	Instructs PRIMOS to name the search list with listname rather than a name derived from pathname. The list name may contain up to 22 characters.
{-NO_SYSTEM | -NS}	Instructs PRIMOS not to preface your search list with the system default rules. If you do not use this option, PRIMOS includes the system default rules at the beginning of your search list.

Search Rule Keywords: Search rule keywords are special instructions that you place in the search rules file. These instructions are carried out when you set the search list or when you perform a search operation on the search list. Keywords enclosed in brackets are variables for which the appropriate literal is supplied when the search list is used.

Keyword	Meaning
-SYSTEM	Allows you to change the placement of system rules in a search list. Specify the -SYSTEM keyword in the desired location in the list. By default, PRIMOS automatically inserts the default search list at the beginning of your file.
-INSERT	Specifies the pathname of another search rules file. PRIMOS inserts the contents of the other search rules file at the point indicated by -INSERT.
-OPTIONAL	Instructs PRIMOS to enable an optional search rule in the search list.
-ADDED_DISKS	Causes PRIMOS to search all of the disk partitions that have been added to your system. This keyword is only used with the ATTACH$ search list.
-STATIC_MODE_LIBRARIES	Instructs PRIMOS to scan all the shared static-mode libraries for the desired entrypoint. This keyword is used only with the ENTRY$ search list.
-PRIMOS_DIRECT_ENTRIES	Instructs PRIMOS to search the PRIMOS system calls. This is only used in the ENTRY$ search list.
[ORIGIN_DIR]	Instructs PRIMOS to search your origin directory. This is useful in completing a pathname in a search rule. This keyword can be used in all search rule files except ATTACH$.

[HOME_DIR] — Instructs PRIMOS to search your home directory (the current attach point, as opposed to the origin directory, which is covered by [ORIGIN_DIR, above). This keyword can be used in all search rule files except ATTACH$. However, be aware that the use of [HOME_DIR] in ENTRY$ can cause unexpected results.

[REFERENCING_DIR] — Instructs PRIMOS to search a pathname that you supply. When the search list is used, the pathname is substituted for the [REFERENCING_DIR] keyword. If the operation that uses the search list does not supply a pathname, PRIMOS ignores the keyword. [REFERENCING_DIR] can be used in all search rule files except ATTACH$.

/*comment_text — Causes PRIMOS to ignore comments beginning with the characters /*.

Uses Of Search Lists

Three ways to use search lists are:

- Use the system default list: In this case, you never need to use SET_SEARCH_RULES or to create your own search list file.
- Use your own list, including the system list: In this case, use SET_SEARCH_RULES without the -NO_SYSTEM option in order to allow PRIMOS to insert the system list at the start of your file.
- Use your own list, overriding the system list: In this case, use SET_SEARCH_RULES with the -NO_SYSTEM option.

The following example shows how to use SET_SEARCH_RULES to change your search rules. In this case, SSR is used to change the command file search rule list from the system default COMMAND$ to MYLIST.COMMAND$.SR. For the sake of the example, only the affected search list, COMMAND$, is displayed:

```
OK, LSR
        .
        .
        .

List: COMMAND$
Pathname of template: <SYSONE>SEARCH_RULES*>COMMAND$

    cmdnc0

        .
        .
        .

OK, SSR MYLIST.COMMAND$.SR -NO_SYSTEM
OK, LSR
        .
        .
        .

List: COMMAND$
Pathname of template: <SYSTWO>JULIAN>MYLIST.COMMAND$.SR

    julian>co.files1
    julian>co.files2
    cmdnc0
    bill>programs

        .
        .
        .

OK,
```

USING THE COPY COMMAND WITH EPFS

Use the COPY command to replace one EPF with another. The format of the command is:

```
COPY source-pathname [target-pathname] [options...]
```

where source-pathname identifies the object you wish to copy (source object), and target-pathname identifies the destination and name of the copied object (target object). (The PRIMOS Commands Reference Guide describes the options to the COPY command.)

If the target object, identified by target-pathname, is open, the COPY command fails, unless the target is an EPF. This example shows what happens when a user tries to copy the file ELEANOR to the open file FRANKLIN:

```
OK, COPY ELEANOR FRANKLIN
"FRANKLIN" already exists, do you wish to overwrite it? YES
File open on delete. Unable to delete file "FRANKLIN" (copy)
ER!
```

Using COPY to Replace an Open EPF File

The COPY command does allow you to specify a target file that is open, provided that the target is an EPF. When you give the command, COPY performs a replacement operation, in two stages:

1. First, COPY changes the name of an open EPF file, which you specify as the target object.
2. COPY then copies the file you specify as the source object into a new file it creates with the original name of the open EPF file.

For this operation to work, the target object must be an EPF, and you must include its .RUN suffix on the command line.

To replace an open EPF file, you give a command like:

```
COPY MYLIB>BETTER_EPF.RUN LIBRARIES*>OLD_EPF.RUN
```

where OLD_EPF is the (possibly open) file you want to replace, and BETTER_EPF is the file you are putting in its place.

In this example, assuming OLD_EPF.RUN is in use, the operation works as follows:

1. The name of the target EPF file is changed. The suffix .RUN is replaced by the suffix .RPn, where n is a digit ranging from 0 through 9. In the above example, OLD_EPF.RUN might be renamed OLD_EPF.RP0.

2. The source EPF file is then copied to target-pathname. In the example, the source file BETTER_EPF.RUN is copied to LIBRARIES*>OLD_EPF.RUN.

By default, COPY tells you that the target EPF file is open, and asks whether or not you want the REPLACE operation completed. COPY also displays the name of the replaced file. These messages are shown in the following example:

```
OK, COPY LIZ>POWER2.RUN LIZ>GO.RUN
EPF file "LIZ>GO.RUN" already exists, do you wish to replace it? YES
New version of EPF file LIZ>GO.RUN now in place.
Old version of active EPF file now named LIZ>GO.RP0.
OK,
```

To prevent the display of these messages, use the -NO_QUERY option of the COPY command.

Once you have replaced the EPF, anyone who invokes it gets the new version. However, if people were using the old version at the time that you replaced it, that version remains mapped into their address spaces. You may want to tell them what you have done, and suggest that they use REMOVE_EPF and invoke the new version if they wish.

REPLACE (.RPn) Files

PRIMOS does not delete REPLACE files when they are no longer mapped into the address space of any users. If you create REPLACE (.RPn) files, then you are responsible for deleting them once they are no longer needed.

Numbering: The suffix numbering sequence starts at .RP0 and continues through .RP9. Thus, there are a maximum of 10 possible REPLACE files. If all possible files exist, PRIMOS asks you whether it can delete one of the REPLACE files that is not currently mapped to any user's address space, as shown in the following example.

```
OK, COPY POWER2.RUN POWERS.RUN
EPF file "POWERS.RUN" already exists, do you wish to replace it? YES
ok to delete EPF file POWERS.RP0? YES
New version of EPF file POWERS.RUN now in place.
Old version of active EPF file now named POWERS.RP0.
OK,
```

Use the -NO_QUERY option to suppress both the prompts shown in the above example.

If all 10 REPLACE (.RPn) files are still mapped into the address spaces of some users, then the COPY operation cannot be completed, as the following example shows.

```
OK, COPY LIZ>CAR.RUN JAMES>CAR.RUN
EPF file "CAR.RUN" already exists, do you wish to replace it? YES
EPF replace files are all in use.
Unable to replace file. "CAR.RUN" (copy)
ER!
```

10

EPFs Calling Programs

From within a program EPF, you may invoke another program in much the same way that you invoke a subroutine. PRIMOS provides two interface subroutines, CP$ and EPF$RUN, that allow a running program to invoke another program or a PRIMOS command.

Using CP$, your EPF can invoke internal PRIMOS commands; external PRIMOS commands; and user-written EPFs, CPL programs, and static-mode programs. Among user-written programs, the EPFs and CPL programs can be invoked in one of three ways: with no information being passed to or from the program (known as program invocation); with information passed to the called program via a command line (known as command invocation); or with information passed via a command line to the called program, and information returned in the form of a text string from the called program to the calling program (known as function invocation).

This chapter:

- Discusses programs, commands, and functions
- Introduces CP$, EPF$RUN, and FRE$RA, the subroutines used for calling and for cleaning up after programs
- Explains how to pass information to and from a called program
- Examines the mechanisms by which programs invoke other programs
- Describes use of the CP$ subroutine

- Explains command preprocessing with CP$
- Explains how to redirect terminal I/O with CP$
- Examines recursive program invocation
- Shows source and output for a sample program

UNDERSTANDING COMMANDS, COMMAND FUNCTIONS, AND PROGRAMS

PRIMOS distinguishes among:

- Internal PRIMOS commands, such as ATTACH, RESUME, ASSIGN, and UNASSIGN
- Internal CPL command functions, such as ATTRIB, WILD, and GVPATH
- External commands, such as COPY, LD, RUNOFF, and SPOOL
- Programs written by users

Internal PRIMOS Commands

PRIMOS contains many internal subroutines, some of which are accessible at command level via command names such as ATTACH and RESUME. These commands are called internal commands. They are not stored on the disk as programs; rather, they reside within PRIMOS itself.

Internal CPL Command Functions

The Command Procedure Language (CPL) within PRIMOS contains internal command functions. Typically, these are accessed from within CPL programs by statements such as:

```
TYPE Your global variable file is [GVPATH].
```

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

External Commands

When you issue a command that is neither an internal PRIMOS command nor an internal CPL command function, PRIMOS looks in the directory CMDNC0 for a program with the same name as the command. Programs in CMDNC0 are called external commands. They are regular programs that have been placed in CMDNC0 by Prime or by your System Administrator.

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

```
RESUME CMDNC0>ED
```

Users' Programs

Whether installed in CMDNC0 as a system-wide command or placed in a user's directory, a program is the basic unit of work under PRIMOS. PRIMOS recognizes three types of programs:

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

You use the RESUME command to invoke these types of programs. To determine what type of program you are invoking, PRIMOS checks the program name's suffix: .RUN for an EPF, .CPL for a CPL program, .SAVE or no suffix for a static-mode program.

Note

SEG programs, which are invoked by the SEG command and use the .SEG suffix, are also static-mode programs.

CP$, EPF$RUN, AND FRE$RA

The two subroutines that allow one program to call another are CP$ and EPF$RUN. Of these, CP$ is the more general. Calling CP$ invokes the PRIMOS command processor. Therefore, you can use CP$ to invoke:

- Internal PRIMOS commands, such as ASSIGN
- External PRIMOS commands
- User-written CPL programs

- User-written EPFs
- User-written static-mode programs

When used to invoke EPFs, CP$ allows you to use such command preprocessing features as wildcarding and name generation.

EPF$RUN, on the other hand, can be used only to invoke EPFs. It does not allow preprocessing. Therefore, you would use EPF$RUN only when you do not want any changes to be made to the command line being passed to the called program.

Information on how to use the CP$ interface is provided later in this chapter and in the Advanced Programmer's Guide, Volume III: Command Environment. Information on how to use the EPF$RUN interface is provided in the Advanced Programmer's Guide, Volume III.

FRE$RA is needed only when you have invoked a program as a function. Such programs allocate memory in which to store a text string that they return to the calling programs. After using the text string, the calling program must call FRE$RA in order to return the storage area to the pool of available memory. For information on FRE$RA, see the Advanced Programmer's Guide, Volume III.

PASSING INFORMATION TO AND FROM PROGRAMS

Three optional parameters, managed by PRIMOS, can be used to communicate with programs:

- A command line
- A severity code
- A returned text string

Command Lines

Any EPF, CPL program, or static-mode program can be written so that it accepts a command line. (In fact, any of these types of programs can be written so that they require command lines; accept them as optional but do not require them; ignore command lines; or reject any attempt to use command lines.)

The command line allows a user to pass any desired operating parameters to the program. For example, a user might invoke a program named MYPROG with the following command:

```
RESUME MYPROG PAYROLL_DATABASE -DELETE_EMPLOYEE 176652
```

In this example, the command line passed to the program MYPROG is:

    PAYROLL_DATABASE -DELETE_EMPLOYEE 176652

The MYPROG program can then choose to parse the passed command line and use it to determine what actions are to be taken.

EPFs: To make an EPF accept a command line, you must have its main entrypoint accept at least two arguments in its calling sequence. The first argument contains the command line as a CHARACTER VARYING string when the EPF is invoked by PRIMOS. The second argument is the severity code.

CPL Programs: To make a CPL program accept a command line, include an &ARGS directive in the program.

Static-mode Programs: To make a static-mode program accept a command line, have it call RDTK$$ to extract the command line from the static command-line buffer before calling COMANL to get a new line of input.

Severity Codes

You may choose to have your program return a severity code indicating the degree to which the program successfully completed its task. The severity code is an integer (16-bit halfword) that is returned by the program to PRIMOS upon completion. PRIMOS then passes the code to the invoker of the program either as an indication on a display (when the invoker is a user) or as a returned value (when the invoker is a program).

Signaling Severity Codes: Successful completion is signaled by returning a severity code of 0 or by choosing not to use the severity code feature at all. A user invoking a program that completes successfully receives the OK, prompt.

Unsuccessful completion is signaled by returning a severity code that is greater than 0. A user invoking a program that completes unsuccessfully receives the ER! prompt.

Successful completion with a warning indicator is signaled by returning a severity code that is less than 0. Warnings are usually signaled by a program when it completes in an orderly fashion, but when it has not

done everything the user expected it to do. A user invoking a program that completes successfully and indicates a warning condition receives the OK, prompt.

Note

The OK, and ER! prompts are the default brief prompts set up upon user login. Users may change these prompts and their corresponding long prompts at will, by using the RDY command, explained in the Prime User's Guide and in the PRIMOS Commands Reference Guide. Among other advantages, customizing prompts in this manner allows users to distinguish between wholly successful completions and successful completions with warning indicators.

Choosing Values for Severity Codes: Although PRIMOS specifies the sign of the severity code (zero, positive, or negative), the programmer decides on the actual values the program will return to indicate an error or warning condition. Typically, the standard PRIMOS error codes, documented in the Subroutines Reference Guide, are used as positive or negative values. However, all severity codes that can be returned by a program should be listed and explained in the documentation for that program.

Returning Severity Codes: To have an EPF return a severity code, have its main entrypoint accept at least two arguments in its calling sequence. The second argument is the severity code, which the EPF sets to some value before returning from the main entrypoint to PRIMOS.

To have a CPL program return a severity code, have it issue a &RETURN directive specifying an integer value for the severity code. For example:

```
&RETURN %SEVERITY_CODE%
```

To have a static-mode program return a severity code, first have it call the SETRC$ subroutine and then have it call the EXIT subroutine. The SETRC$ subroutine is described in the Subroutines Reference Guide.

Returned Text Strings

Programs designed as functions return text strings as the result of their invocation. For example, a program named USER_ID might return the username of the invoking user; it could be used as a function in a CPL program as follows:

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

Programs designed as functions must be EPFs or CPL programs; static-mode programs cannot return text strings.

EPFs: To have an EPF return a text string, have its main entrypoint accept eight arguments. These arguments, and their functions, are explained in the Advanced Programmer's Guide, Volume III: Command Environment.

Using these arguments, the EPF can allocate temporary memory storage for the text string to be returned, copy the text string into that storage, store the pointer to the allocated storage as an argument to be returned to its caller, and return from its main entrypoint to PRIMOS.

After the calling program makes use of the returned string, it must return the allocated storage that contained the returned text string to the pool of available memory. To return this storage, it must use the FRE$RA subroutine, as explained later in this chapter.

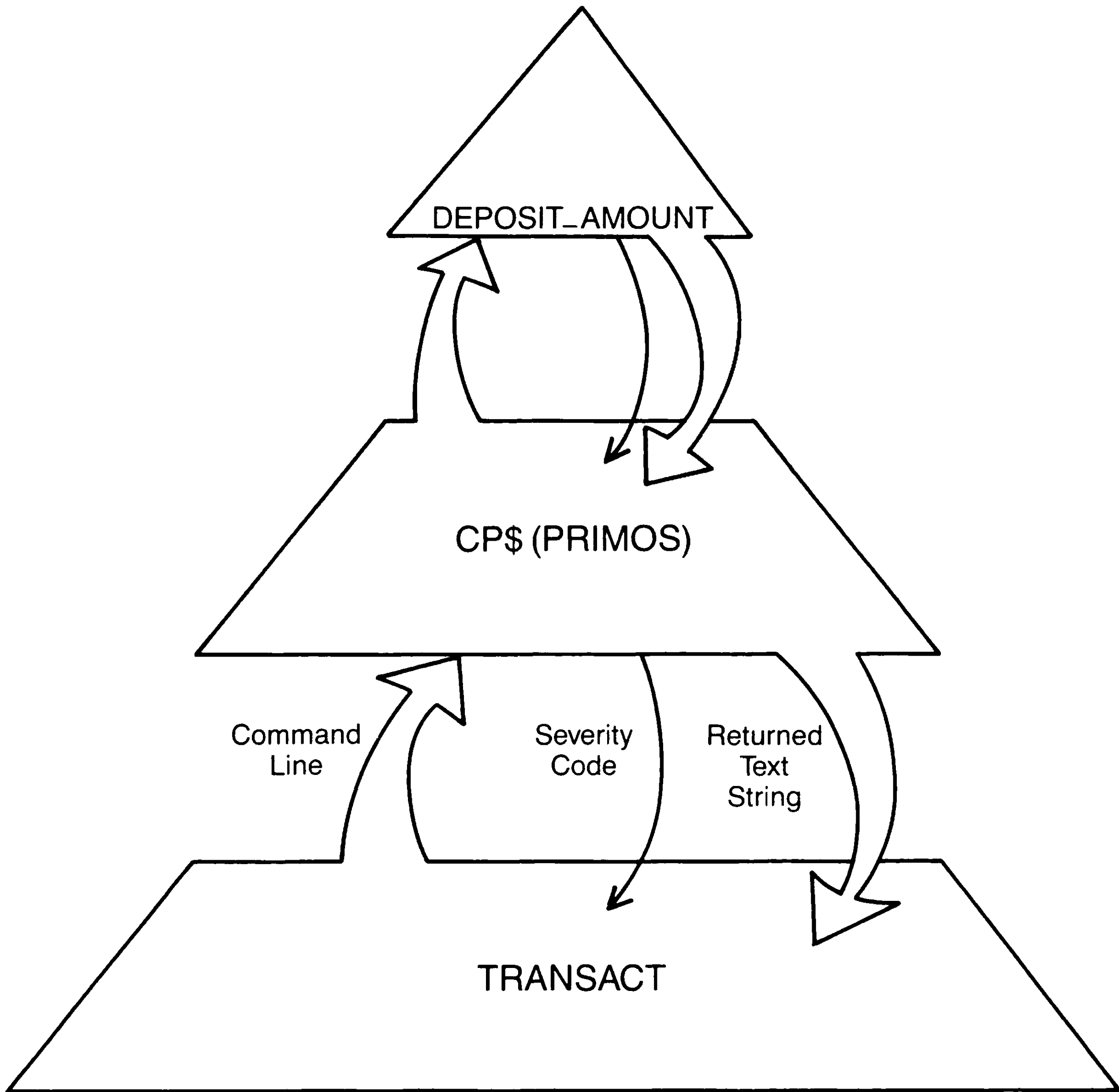
CPL Programs: To have a CPL program return a text string, have it issue a &RESULT directive before it issues the &RETURN directive. For example:

```
&RESULT [CALC %HORNS% / 2]  /* Return number of steers.
&RETURN
```

UNDERSTANDING PROGRAM INVOCATION

As the preceding discussion of argument passing suggests, invoking a command or program from within another program is similar to calling a subroutine. Passing parameters to a program is done by passing textual information on a command line. If the called program is to return a value to the calling program, the called program is designed as a function and returns the value as a textual function value.

Figure 10-1 illustrates the effect of one program named TRANSACT calling another program named DEPOSIT_AMOUNT. The example uses the CP$ interface. It shows the layer of PRIMOS that is invoked when CP$ is called by TRANSACT and that ultimately calls DEPOSIT_AMOUNT. Later, when the DEPOSIT_AMOUNT program finishes, it returns to PRIMOS. PRIMOS then performs some cleanup activities, and then returns from CP$ to the TRANSACT program, resuming its execution.



Invocation of One Program by Another Using CP$
Figure 10-1

Use of Shared Memory

A major difference between programs invoking programs and programs invoking subroutines lies in the use of shared memory. Calling programs and their target subroutines often expect to share information through a COMMON or STATIC EXTERNAL area in memory. This type of sharing cannot be done when one program invokes another, because PRIMOS keeps individual program invocations entirely separate.

Note

If it is absolutely necessary to employ a common area of memory for use by multiple program invocations, you must use the SYMBOL subcommand of BIND to specify where a COMMON or STATIC EXTERNAL area is to be placed in memory. The same SYMBOL command must be used in linking each program that shares the area. This functionality is rarely needed.

Limits on Program Invocation

There are resource limits on the invocation of programs from within programs. Limits exist on:

- The maximum number of programs at a given command level (program breadth)
- The maximum number of dynamic segments
- Memory utilization

The System Administrator at each site sets these limits, which are discussed in Chapter 7 of this book.

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 the Advanced Programmer's Guide, Volume III: Command Environment. In summary, the interface has three 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 command function, and it ignores any command line passed to it. No severity code is returned; therefore, 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 command function.

3. Function invocation. The program being invoked accepts:

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

   As with command invocation, a program that is a command function returns a severity code.

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

Program Invocation: The main entrypoint of a program must take no arguments. In FORTRAN, the main entrypoint of a program named MYPROG begins as follows:

```
SUBROUTINE MYPROG
```

In PL1/G, the same entrypoint would begin with:

```
myprog: proc;
```

Command Invocation: The main entrypoint of a command takes two arguments. In FORTRAN, the main entrypoint of a command named MYPROG begins as follows:

```
SUBROUTINE MYPROG(CMDLIN,CODE)
INTEGER*2 CMDLIN(513),CODE
```

A PL1/G command named YOURPROG begins as follows:

```
yourprog: proc(command_line,code);

dcl command_line char(1024) var,
    code fixed bin(15);
```

In each case, the command line is passed as a varying character string. The program parses the command line, using a subroutine such as CMDL$A or CL$PIX.

Before executing a RETURN statement, the program should set CODE to the final severity code for the command execution. A value of 0 indicates successful completion; a value of > 0 indicates an unsuccessful completion; and a value of < 0 indicates a successful completion with a warning condition.

Function Invocation: The main entrypoint of a function takes five arguments. This chapter discusses how to invoke a function and accept the returned value. For information on how to write a command function, see the Advanced Programmer's Guide, Volume III: Command Environment.

USING THE CP$ SUBROUTINE

There are three uses for CP$:

- Invoking internal PRIMOS commands
- Invoking commands or programs
- Invoking functions

Each use is described below.

Using CP$ to Invoke an Internal PRIMOS Command

To use the CP$ subroutine to invoke an internal PRIMOS command, use the following calling sequence:

```
dcl cp$ entry(char(1024) var,fixed bin(15),fixed bin(15));

call cp$(command_line,code,command_status);
```

When you call CP$, the command processor attempts to execute the internal 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 command_status.

Possible values for command_status vary, but they are almost always 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

The returned value of command_status is undefined if the returned value of code is non-zero.

The Form of the Command Line: In command_line, 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 use by a running program, you might have the program call CP$ with the following command line:

```
ASSIGN MT0 -WAIT
```

The RESUME command is a special case, because it runs an external program. Use of the RESUME command to invoke a program via CP$ is discussed in the next section.

Using CP$ to Invoke a Command or a Program

To use the CP$ subroutine to invoke a command or a program, use the following calling sequence. (This sequence is the same one used for invoking internal commands.)

```
dcl cp$ entry(char(1024) var,fixed bin(15),fixed bin(15));

call cp$(command_line,code,command_status);
```

When you call CP$, the command processor attempts to execute the command line 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 command_status.

Note

The returned value of command_status is undefined if the returned value of code is non-zero.

The Form of the Command Line: When calling user-written programs (other than those that have been placed in CMDNC0), the text string passed in command_line begins with:

```
RESUME program-pathname
```

The RESUME command is an internal PRIMOS command, and program-pathname is the pathname of the program you wish to invoke. If there are any command line parameters, they must follow program-pathname in command_line.

When invoking external PRIMOS commands (that is, programs that are stored in CMDNC0), command_line begins with:

```
command-name
```

This string invokes the external command named command-name.

Using CP$ to Invoke a Function

The CP$ subroutine can invoke either an internal PRIMOS command function or a user-written function.

To use the CP$ subroutine to invoke a function, use the following calling sequence:

```
dcl cp$ entry(char(1024) var,fixed bin(15),fixed bin(15),
           1, 2 bit(1),
              2 bit(1),
              2 bit(14),
           ptr,ptr);

/* Preset values returned by invoked command to null values,
   in case command does not set them or does not in fact get
   invoked, so that we use legitimate default values. */

rtn_function_ptr=null();
command_status=0;

call cp$(command_line,code,command_status,command_flags,
     null(),rtn_function_ptr);
```

When you call CP$, the command processor attempts to execute the command line 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 command_status.

The command_flags structure (shown below) specifies that the caller expects a string to be returned. null() is the null pointer; it generates a pointer of 7777/0.

A pointer to the returned text string structure is returned in rtn_function_ptr if the invoked program has successfully set its return value. If rtn_function_ptr is still null() when CP$ returns, no text string has been returned.

Note

The returned values of command_status and rtn_function_ptr are undefined if the returned value of code is non-zero.

Setting the Command Flags: Declare and pass the following PL1/G structure in command_flags:

```
dcl 1 command_flags static,
      2 command_function_call bit(1) init('1'b),
      2 no_eval_vbl_fcn_refs bit(1) init('0'b),
      2 mbz bit(14) init('00000000000000'b);
```

Setting the command_function_call bit to '1'b announces that the caller expects a text string to be returned by the called function. Setting the no_eval_vbl_fcn_refs bit to '0'b allows the command processor to evaluate variable and function references in the command line. The mbz bits must always be set to zero.

The Form of the Command Line: Use the command function name (for a PRIMOS command function) or the RESUME command followed by the name of the function (for a user-written function), just as you would when invoking a command. Do not enclose the command line in square brackets ([ ]).

For example, to identify the user's global variables file, call CP$ with the command line:

```
GVPATH
```

The call itself is:

```
call CP$('GVPATH',code,command_status,command_flags,
        null(), rtn_function_ptr);
```

This call returns either the pathname of the global variables file or the keyword -OFF in a structure pointed to by rtn_function_ptr. This structure is described in the next section.

To invoke a user-written function, call CP$ with a command line such as the following:

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

The calling sequence for this call is:

```
call CP$('RESUME PROGRAMS>GET_RECORD 1154 -DATABASE PAYROLL',
        code,command_status,command_flags,null(),
        rtn_function_ptr);
```

Again, information is returned by the function in a structure pointed to by rtn_function_ptr.

The Returned Text String: The returned text string is the returned value of the function.

The declaration of the returned text string structure is:

```
dcl 1 rtn_function_structure based(rtn_function_ptr),
      2 version fixed bin(15),
      2 text_string char(32767) var;
```

Notes

If the invoked function did not return a value, then rtn_function_ptr is not modified. Therefore, set it to the null pointer before calling CP$, and check it after CP$ returns to be sure that a result has been returned.

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

Freeing Storage: After using the returned text string, your program should free the returned text string structure to the pool of available memory. Use FRE$RA to do this, as follows:

```
dcl fre$ra entry(ptr);

call fre$ra(rtn_function_ptr);
```

Note

Do not call FRE$RA with a null pointer.

Command Preprocessing

When invoking its target program, the CP$ subroutine performs any appropriate command preprocessing. Command preprocessing can include:

1. Multiple command processing (such as ATTACH MYDIR; LD)
2. Variable and function evaluation (such as %MYVAR% and [GVPATH])
3. Iteration (such as DELETE MYPROG.(FTN BIN LIST RUN))
4. Treewalking (such as SIZE *>@@>MYPROG.PL1G)
5. Wildcarding and name generation (such as CMPF *>OLD>@@ *>NEW>==)

These features are summarized in the Prime User's Guide and described in detail in the PRIMOS Commands Reference Guide.

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 substituted in the command line.

Preprocessing for Commands: The command separator character, which is the semicolon (;), indicates that multiple commands are present. The command processor separates each command, expands variable and function references (unless inhibited), and then determines further preprocessing by analyzing each resulting command.

CP$ determines the appropriate remaining command preprocessing features by checking the command type (internal CPL program, EPF, or static-mode program).

If the command being invoked is an internal command, the remaining preprocessing depends on the command itself, based on a table inside PRIMOS. Otherwise, the command invokes a program.

If the program being invoked is a CPL program, CP$ performs only the iteration function. Treewalking, wildcards, and name generation names are passed to the CPL program untouched.

If the program being invoked is an EPF, CP$ uses the information provided in the EPF by BIND during program linking to determine which command preprocessing functions are to be performed. (Chapter 8 contains information on the BIND subcommands that govern command preprocessing for an EPF.)

If the program being invoked is a static-mode program, CP$ uses the name of the program to determine which command preprocessing functions are to be performed. If the name begins with NX$, then no additional command preprocessing functions are performed. If the name begins with NW$, then iteration is performed, but not treewalking, wildcarding, or name generation. If the name does not begin with NX$ or NW$, then all command preprocessing functions are performed.

Preprocessing for Functions: No command preprocessing is performed when CP$ invokes a function, except for variable and function evaluation (unless inhibited). Because a function must return a single value, features such as multiple commands or iteration are not appropriate when invoking a function.

Terminal Input and Output

When you invoke a command or a program from within another program, all output from the called program is sent, by default, to the user terminal. Similarly, responses to prompts or queries issued by the command are sought, by default, from the user terminal.

If you do not want to require terminal interaction, you can redirect the called program's output into a command output file. You can also have the called program seek its input from a command input file.

You can use either the COMO$$ subroutine or the internal PRIMOS command COMOUTPUT to redirect terminal output to a command output file. For example, to direct output from the LD command to a command output file named LD.COMO, you might use the following pair of calls:

```
call CP$('COMOUTPUT LD.COMO',code,command_status)
     /* Open the comoutput file */
call CP$('LD MYDIR -DETAIL -NO_WAIT',code,command_status)
     /* Collect the information. The -NO_WAIT option
        suppresses LD's --More-- prompt */
call CP$('COMOUTPUT -END',code,command_status)
     /* Close the comoutput file */
```

To have the called program receive input from a command input file, use either COMI$$ or the internal PRIMOS command COMINPUT.

Error Codes From CP$

An output argument, code, informs the calling program of the success or failure of the operation. This argument is an INTEGER*2 or FIXED BIN(15) variable. Symbols are provided to allow PL1/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 an EPF, either via the RESUME command or by specifying an EPF in CMDNC0, an error code returned by the EPF$RUN subroutine is returned by CP$. Therefore, consult the list of error codes returned by EPF$RUN, given in the Advanced Programmer's Guide, Volume III: Command Environment, for information on additional error codes returnable by CP$.

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.

Keyword	Value	Meaning
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. Either you do not have access to the program or the program does not exist.

IF A PROGRAM INVOKES ITSELF

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

A program invoking itself recursively via CP$ or EPF$RUN, whether directly or indirectly, does not necessarily produce the same results as if it calls itself by invoking its own main entrypoint. The first process uses recursive program invocation, and the second uses recursive procedure invocation. Either of these processes causes dynamic storage to be allocated and initialized for each invocation. However, only program invocation causes PRIMOS to allocate and initialize static storage. Each time the program is invoked, static storage is allocated for all procedures in that program. Once the program is running, no additional static storage is allocated by PRIMOS.

To see how this might affect a program's output, consider the following PL1/G program:

```
divine: proc(command_line,code);

dcl command_line char(1024) var,
    code fixed bin(15);

dcl number fixed bin(15),
    sum fixed bin(15) static initial(0);

number=bin(command_line);  /* Get the number. */
if number=10 then return;  /* At 10, we stop. */

sum=sum+number;  /* Add number into sum. */

put list('Invocation number '||trim(char(number),'11'b)||
         ', sum is now '||trim(char(sum),'11'b));
put skip;

call divine(trim(char(number+1),'11'b),code);  /* Call myself. */

put list('Finishing invocation #'||trim(char(number),'11'b)||
         ', sum is now '||trim(char(sum),'11'b));
put skip;

code=0;  /* Always successful. */
return;  /* Finished. */

end;
```

When the above program is compiled, linked, and executed, it produces the following output:

```
Invocation number 0, sum is now 0
Invocation number 1, sum is now 1
Invocation number 2, sum is now 3
Invocation number 3, sum is now 6
Invocation number 4, sum is now 10
Invocation number 5, sum is now 15
Invocation number 6, sum is now 21
Invocation number 7, sum is now 28
Invocation number 8, sum is now 36
Invocation number 9, sum is now 45
Finishing invocation #9, sum is now 45
Finishing invocation #8, sum is now 45
Finishing invocation #7, sum is now 45
Finishing invocation #6, sum is now 45
Finishing invocation #5, sum is now 45
Finishing invocation #4, sum is now 45
Finishing invocation #3, sum is now 45
Finishing invocation #2, sum is now 45
Finishing invocation #1, sum is now 45
Finishing invocation #0, sum is now 45
OK,
```

In this example, procedure DIVINE invokes itself ten times within a single program invocation. The static storage is initialized only once, when the RESUME DIVINE command is issued. This storage is then reused by each invocation of the procedure DIVINE. Each invocation of the procedure changes the contents of the static storage used by SUM. The change is visible across all procedure invocations.

Now, the program is modified to call itself by using CP$ instead of by invoking its main entrypoint. The program now reads:

```
divine: proc(command_line,code);

dcl command_line char(1024) var,
    code fixed bin(15);

dcl number fixed bin(15),
    sum fixed bin(15) static initial(0),
    status fixed bin(15);

dcl cp$ entry(char(1024) var,fixed bin(15),fixed bin(15));

number=bin(command_line);  /* Get the number. */
if number=10 then return;  /* At 10, we stop. */

sum=sum+number;  /* Add number into sum. */
```

```
put list('Invocation number '||trim(char(number),'ll'b)||
        ', sum is now '||trim(char(sum),'ll'b));
put skip;

call cp$('RESUME DIVINE '||trim(char(number+1),'ll'b),status,
      code);  /* Call myself. */

put list('Finishing invocation #'||trim(char(number),'ll'b)||
        ', sum is now '||trim(char(sum),'ll'b));
put skip;

if status^=0 then code=status;  /* Invocation error. */
   else code=0;  /* Always successful if invoked. */
return;  /* Finished. */

end;
```

When the above program is invoked, the output is:

```
Invocation number 0, sum is now 0
Invocation number 1, sum is now 1
Invocation number 2, sum is now 2
Invocation number 3, sum is now 3
Invocation number 4, sum is now 4
Invocation number 5, sum is now 5
Invocation number 6, sum is now 6
Invocation number 7, sum is now 7
Invocation number 8, sum is now 8
Invocation number 9, sum is now 9
Finishing invocation #9, sum is now 9
Finishing invocation #8, sum is now 8
Finishing invocation #7, sum is now 7
Finishing invocation #6, sum is now 6
Finishing invocation #5, sum is now 5
Finishing invocation #4, sum is now 4
Finishing invocation #3, sum is now 3
Finishing invocation #2, sum is now 2
Finishing invocation #1, sum is now 1
Finishing invocation #0, sum is now 0
OK,
```

In this example, it is the program named DIVINE that invokes itself ten times. Each time the RESUME DIVINE command is issued, once by the user and ten times by succeeding invocations of the program, PRIMOS allocates and initializes static storage for the program. This static storage includes the variable SUM. Therefore, each invocation of DIVINE starts off with SUM set to 0. Each invocation of DIVINE sets SUM to its invocation number. In doing so, however, it modifies only its own copy of SUM. As seen by the final messages, copies of SUM for other invocations are not changed.

The point to remember, therefore, is that 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 with the SYMBOL subcommand of BIND. Because PRIMOS separates program invocations so that they do not destroy one another's data, one program can be invoked and then suspended; 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 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 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.

SAMPLE PROGRAM

This sample program, named DISPLAY_RESULT, accepts a command line containing name and arguments for a command function. It then invokes the function and displays whatever text string the function returns. You might use this program to test a command function you were developing.

If no text string is returned to DISPLAY_RESULT, either because no command was invoked or because the command invoked was not a function, DISPLAY_RESULT returns the message, "No result returned."

Source code for DISPLAY_RESULT is:

```
display_result: proc(command_line,code);

dcl command_line char(1024) var,
    code fixed bin(15);

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

dcl command_status fixed bin(15),
    1 command_flags static,
      2 command_function_call bit(1) init('1'b),
      2 no_eval_vbl_fcn_refs bit(1) init('0'b),
      2 mbz bit(14) init('00000000000000'b),
    rtn_function_ptr ptr;

dcl 1 rtn_function_structure based(rtn_function_ptr),
      2 version fixed bin(15),
```

```
      2 text_string char(32767) var;

dcl cp$ entry(char(1024) var,fixed bin(15),fixed bin(15),
              1, 2 bit(1),
                 2 bit(1),
                 2 bit(14),
              ptr,ptr),
    errpr$ entry(fixed bin(15),fixed bin(15),char(80),
                 fixed bin(15),char(80),fixed bin(15)),
    fre$ra entry(ptr);

/* Preset command_status and rtn_function_ptr to null values, in
   case they are not set by the invoked program. */

rtn_function_ptr=null();
command_status=0;

/* Now call CP$ with the exact command line passed. */

call cp$(command_line,code,command_status,command_flags,
      null(),rtn_function_ptr);
if code^=0
   then do;
        call errpr$(k$irtn,code,'Cannot invoke program',21,
              'DISPLAY_RESULT',14);
        return;
        end;  /* if code^=0 */

code=command_status;  /* Pass command status on. */

if rtn_function_ptr=null()
   then do;
        put list('No result returned.');
        put skip;
        end;  /* if rtn_function_ptr=null() */

   else do;  /* if rtn_function_ptr^=null() */

        if rtn_function_ptr->version=0
           then do;
                put list(rtn_function_ptr->text_string);
                put skip;
                end;  /* if version=0 */

           else do;  /* if version^=0 */

                put list('Bad version, is '||trim(char(
                       rtn_function_ptr->version),'11'b)||
                       ', should be 0.');
                put skip;

                code=e$bver;

                end;  /* if version^=0 */
```

```
        call fre$ra(rtn_function_ptr);  /* Free text. */

        end;  /* if rtn_function_ptr^=null() */

    end;  /* display_result: proc */
```

A sample session involving this program, and a discussion of that session, follow.

```
OK, PL1G DISPLAY_RESULT
[PL1G Rev. 19.4]
0000 ERRORS [PL1G Rev. 19.4]
OK, BIND
[BIND rev 19.4]
: LOAD DISPLAY_RESULT
: LIBRARY PL1GLB
: LIBRARY
BIND COMPLETE
: FILE
OK, RESUME DISPLAY_RESULT                                                    I
No result returned.
OK, RESUME DISPLAY_RESULT DATE
84-11-13
OK, RESUME DISPLAY_RESULT RDY
OK 11:41:00  8.330  16.724
No result returned.
OK, RESUME DISPLAY_RESULT ATTRIB DISPLAY_RESULT.PL1G -LENGTH
912
OK, RESUME DISPLAY_RESULT TYPE FOO
FOO
No result returned.
OK, RESUME DISPLAY_RESULT COMPARE_FILE
ER! RESUME DISPLAY_RESULT COMPARE_FILE LOGIN.CPL LOGIN.CPL                   II
TRUE
OK, RESUME DISPLAY_RESULT COMPARE_FILE LOGIN.CPL MEMO_TO_CHARITY
FALSE
OK, ASSIGN MT(0 1 2)
Too many objects specified. 1 (asnmt$)
USAGE: ASSIGN MTn [-ALIAS MTm] [<options>]
       ASSIGN MTX  -ALIAS MTn  [<options>]
        options: [ -WAIT ]
                 [ -MOUNT ]
                 [ -RETENSION ]
                 [ -TPID <id> ]                                              III
                 [ -7TRK | -9TRK ]
                 [ -SPEED {25 | 100} ]
                 [ -RINGON | -RINGOFF ]
                 [ -DENSITY {800 | 1600 | 3200 | 6250} ]
ER! RESUME DISPLAY_RESULT ASSIGN MT(0 1 2)
Device MT0 assigned.
No result returned.
```

```
Device MT1 assigned.
No result returned.
Device MT2 assigned.
No result returned.
OK, UNASSIGN MT(0 1 2)
Too many objects specified. 1 (usnmt$)
Usage: UNASSIGN MTn [-UNLOAD]
       UNASSIGN -ALIAS MTn  [-UNLOAD]
ER! RESUME DISPLAY_RESULT UNASSIGN MT(0 1 2)
Device released.
No result returned.
Device released.
No result returned.
Device released.
No result returned.
OK,
```

III

For purposes of analysis, this session may be divided into three parts:

Part I: The user invokes DISPLAY_RESULT with no argument, and gets the appropriate message in return. The user then invokes DISPLAY_RESULT to invoke internal commands (RDY and TYPE) and two internal command functions (ATTRIB and DATE). Note that the commands RDY and TYPE display their own results at the user's terminal, but do not return text strings to DISPLAY_RESULT. Thus, invoking these internal commands causes the message printed by those commands to be displayed first, followed by the message "No result returned" from DISPLAY_RESULT.

Part II: The user invokes a program named COMPARE_FILE, which resides in CMDNC0. This program is not a Prime program, but is an example of how a program may be installed by the System Administrator as a new command and subsequently used by all users. COMPARE_FILE is a command function that compares two files for equality. If the files are identical, COMPARE_FILE returns the text string "TRUE". If the files are not identical, it returns "FALSE". If it encounters an error, it returns a positive error code. Under no circumstances does it display any messages to the terminal. DISPLAY_RESULT assumes that the command it invokes will display an error message before returning a positive severity code. Therefore, when DISPLAY_RESULT is used to invoke COMPARE_FILE, and COMPARE_FILE encounters an error, no message is displayed except for the ER! prompt. (An error is shown in the case when two pathnames are not supplied to COMPARE_FILE.)

Part III: This part of the session demonstrates the subtle differences in command preprocessing between invoking internal PRIMOS commands and invoking external programs. As the session shows, the ASSIGN and UNASSIGN commands do not accept iteration. However, the DISPLAY_RESULT program does accept iteration. Therefore, using DISPLAY_RESULT with iteration to invoke ASSIGN and UNASSIGN causes DISPLAY_RESULT itself to be invoked several times, each time invoking ASSIGN or UNASSIGN with a single device name.

APPENDIXES

A
BIND Error Messages

This appendix lists (in alphabetical order) the error messages that you may receive when using BIND.

- Attempt to initialize dynamic common

This is an internal error, usually involving a CC or VRPG program. Ensure that your program compiles correctly and retry the BIND session. If the error persists, contact your Customer Support Center.

- :attempt to reference undefined common

A common area is referenced in a module, but the area has not been defined or loaded. This error may also result from an internal error in a user-written compiler.

- Bad address format. Correct syntax is <octal>/<octal>.

An incorrect absolute address format is given. Usually the slash separator has been omitted.

- :bad command

You issued an invalid or a misspelled subcommand. The subcommand is ignored.

- Bad EPF. It will not be saved.

You attempted to save an EPF that contains nothing.

- Bad group type

The object file does not meet BIND's expectations. This message usually results from an internal error in a user-written compiler. Make sure that all of your program modules compile with no errors. If the message persists, call your Customer Support Center.

- Bad object file

You tried to link a file that has faulty code. Either the file is not an object file or it has been compiled incorrectly.

- Bad size on allocate

The size argument supplied by the user for the ALLOCATE command is missing, or is not a number, or is a negative number.

- :bad symbol name

The symbol name supplied by the user for the ALLOCATE command begins with a number or hyphen, which constitute illegal symbol names.

- Base area zero full

All locations in the sector zero area have been used. The last linked binary file has done too many indirect references and needs to be broken down into smaller modules.

- BIND can't link this binary file

A special object group was generated by a compiler which specified which linkers could be used to link the binary file. In this case, the BIND bit was not set.

- BIND internal error. Linkage table corrupted.

BIND's internal linkage descriptors have been corrupted. If the message persists, call your Customer Support Center.

- Block size mismatch

This is an internal error message. This may point to an incorrect binary file format. Be sure that all of your programs compile with no errors. If the message persists, call your Customer Support Center.

- Can't load EPF

You tried to link an EPF runfile after you linked one or more modules. In a BIND session, the EPF runfile must be the first module linked.

- Can't load in 32r mode

You tried to link an object module compiled or assembled in 32R mode. Recompile with the -64V option or reassemble in V mode or I mode.

- Can't load in sectored mode

You attempted to link an object module compiled or assembled in S mode, an archaic mode provided only for compatibility with Honeywell 316 and 516 computers. Recompile with the -64V option or reassemble in V mode or I mode.

- :common redefined as ECB entry.

The symbol was first defined as a common definition and then defined as an ECB entry.

- Debug group encountered before a procedure definition group

A compiler emitted DBG information before the procedure section of the program. Be sure that all of your modules compile with no errors. If the message persists, call your Customer Support Center.

- :duplicate option

The desired option to a subcommand was given more than once on the same command line.

- :ECB entry redefined as common.

The symbol was first defined as an ECB entry and then defined as a common definition.

- :ecb not found

An attempt is made to define as an ECB a module that has not been linked prior to using the MAIN subcommand.

- ** Empty file **

You tried to link an empty object file.

- External memory reference to illegal segment

A compiler tried to place a COMMON block in an inappropriate segment. This is an internal error. Be sure that all modules compile with no errors. If the message persists, call your Customer Support Center.

- :extra map destination. First one used.

You specified too many map destinations. The first map destination given is used.

- :extra map option. First one used.

You specified too many map options. The first map option given is used.

- Illegal addressing mode

A direct reference has been made to common areas located in another segment. Try to recompile.

- Illegal block type

This is a compiler-generated message. Be sure that all modules compile with no errors. If the message persists, call your Customer Support Center.

- :illegal indirect or index on an address constant

This is usually generated by the compiler or the assembly code. This indicates that there is an incorrect instruction format or that the index is out of range. Be sure that all modules compile with no errors. If the message persists, call your Customer Support Center.

- :illegal redefinition of common

A common area has been defined by a module to be a certain size and subsequently redefined to be larger. This can happen when a module is placed in memory via the RESOLVE_DEFERRED_COMMON or ALLOCATE subcommand, or when data is allocated from the program (for example, the INIT statement in PL/I). This message is a warning message if the commmon area has not been allocated. Otherwise, this message is an error message; the attempt to redefine the common area to be larger is ignored, and the module that made the attempt is not linked.

- Is not a dam or cam file

You tried to file an EPF into a file that is not a DAM or CAM file.

- Is not an ecb

You defined as the MAIN entrypoint a symbol that is not the ECB for a subroutine.

- :Larger redefinition of common.

You redefined a deferred COMMON block to be larger than a previous definition. The size becomes what is specified in the new definition. To inhibit this message, use the NO_COMMON_WARNING subcommand.

- Missing parameter

The parameter following a subcommand is missing, misspelled, or incorrect.

- Missing procedure end group

This is a compiler-generated message. Be sure that all modules compile with no errors. If the message persists, call your Customer Support Center.

- :multiple indirect

This message may be caused by a program compiled in 32R mode. Recompile with the -64V option or reassemble in V mode or I mode.

- Must enter libmode command before entryname command

An attempt was made to create a table of entrypoints without first designating a library class.

- No library mode given

You tried to create a library without designating the library class.

- Not enough storage.

There are no more dynamic segments. Try to clean up your environment (for example, with the REMOVE_EPF and RELEASE_LEVEL commands).

- Old object file

The object file was compiled or assembled in an old version of the compiler or assembler. Recompile or reassemble and restart the BIND session.

- Procedure size too large

Most often this is a compile time error message. This is an added check in the linker that indicates that the code for one procedure and its base areas cannot fit into one segment. Be aware that PMA will not trap this condition at assembly time.

- Runfile too old. Can't reload.

An attempt was made to read a pre-Rev. 19.4 EPF.

- :smaller redefinition of common

You redefined a COMMON block from a larger size to a smaller size. The larger size is used. To inhibit this warning message, issue the NO_COMMON_WARNING subcommand.

- :symbol already exists

You tried to define a new symbol, but the symbol is already in the symbol table. You must give the symbol a new name.

- :symbol not found

You tried to use a symbol with a name that has not been defined.

- This binary requires a newer version of BIND

A special object group was generated by a compiler which specified that a certain version of a linker was required to link the binary file.

- :undefined segment

An attempt was made to access a segment that is not available. This message can come from the compiler or from BIND. It can also mean that you attempted to initialize a static segment greater than one segment. If the message persists, call your Customer Support Center.

- :undefined symbol

An attempt was made to equate two symbols, but the first symbol is not in the symbol table.

- :unrecognized keyword

You used a subcommand with a keyword that is not recognized by BIND.

- :unrecognized option

You used a subcommand with an option that is not recognized by BIND.

- Unsupported segment type

This is a BIND internal error. Recompile with no errors and rebind. If the message persists, call your Customer Support Center.

- Warning: BIND not complete

There still exist unresolved references (that is, calls to subroutines that are not in the runfile). Make sure that all user subroutines and Prime special libraries have been loaded. Also, make sure that any calling program is linked before the subroutines it calls.

- Warning: External reference to DYNT.

A reference was made to an external object, but that object had already been defined as a dynamic entry.

- Warning: Initialization of static common ignored.

An attempt was made to initialize static common (common placed via the SYMBOL subcommand).

B

EPF Error Messages

This appendix lists the EPF-specific error messages that you may encounter while running an EPF.

User-rectifiable Error

- You have exceeded your maximum number of command levels.

This message means that you have reached mini-command level, described in Chapter 7.

Messages for Your System Administrator

All the remaining error messages include the words "Please contact your system administrator." If you encounter any of these errors, record the message and report it to your System Administrator, who should contact your Customer Support Center.

- An error was encountered while attempting to de-allocate EPF procedure segments for EPF-name. Please contact your system administrator.

You may run into this problem as follows: when you use RELEASE, REMOVE_EPF, or INITIALIZE_COMMAND_ENVIRONMENT; or, when the program you are running requires PRIMOS to deallocate a procedure segment.

- Internal EPF error: the EPF level cache has become circularized. Please contact your system administrator.

The EPF level cache contains a linked list of objects. A list is said to be circularized when the end of the list points back to the beginning.

- Internal EPF error: the list of active EPFs for this process has become circularized. Please contact your system administrator.

PRIMOS maintains a list of active EPFs for each running process; this list has been corrupted.

- Internal EPF error: the ENTRY$ search list for this process has become circularized. Please contact your system administrator.

This error can occur only when you are running a program that uses your search list. (Search lists are discussed in Chapter 9.)

- Internal EPF error: the segment mapping table for EPF file EPF-name could not be retrieved. Please contact your system administrator.

This message indicates a problem with storage allocation. One possible cause is that a user might have inadvertently written data into a part of memory used by PRIMOS.

- Internal dynamic storage error: the EPF dynamic storage class has been corrupted. Please contact your system administrator.

The storage class used by PRIMOS to track EPF's use of dynamic segments is not working properly. (A storage class is a heap, that is, a unit of storage managed as a pile so that information can be taken from any part of it.)

- Internal dynamic storage error: the USER dynamic storage class has been corrupted. Please contact your system administrator.

The storage class, or heap, used by PRIMOS to track an individual's use of dynamic segments is not working properly.

- Internal EPF error: the depth of the program session for this user is invalid. Please contact your system administrator.

The command environment breadth is an invalid number (less than 0). This is an internal error, although it may be the result of an errant program writing on memory used by PRIMOS. If the error persists, contact your Customer Support Center.

- Internal EPF error: the command environment level structure is invalid. Please contact your system administrator.

This is an internal error, although it may be the result of an errant program writing on memory used by PRIMOS. If the error persists, contact your Customer Support Center.

- Internal EPF error: the EPF segment mapping table for EPF-name has become invalid. Please contact your system administrator.

The table that PRIMOS uses to track the mapping of segments for this program is not working properly.

C

Glossary

abbreviation
(1) A system-defined short name that can be substituted for a PRIMOS internal command. For example, LE is an abbreviation for the LIST_EPF command. (2) A user-defined short name that can be substituted for one or more PRIMOS commands. A user-defined abbreviation is created with the PRIMOS command ABBREV.

absolute address
An address in virtual memory that, once assigned, cannot be relocated or dynamically assigned. To prevent ambiguity, it may also be called an actual address (as the term "absolute address" may also apply to the physical memory address following virtual memory translation). Use the SYMBOL subcommand of BIND to set an absolute address.

absolute segment number
A segment number that is assigned by BIND to the same virtual segment number. It cannot be relocated or dynamically assigned. It may also be called an actual segment number.

access rights
The ways in which a user may access a file, directory, or segment. Access rights to segments are discussed in Chapter 9. Access Control Lists (ACLs) control access to file system objects and are discussed in the Prime User's Guide.

actual address
See absolute address.

actual segment number
See absolute segment number.

address space
Memory space, usually the memory allocated to one user.

allocate
To reserve an amount of memory for use by a program or process.

allocated storage
Memory space that has been reserved.

assigned to
An EPF is assigned to a segment if PRIMOS gives it addresses within that segment at runtime.

binary file
A file in machine language produced from source code by a compiler, interpreter, or assembler. Binary files must be linked into runfiles by BIND, SEG, or LOAD. Also called an "object module," "object file," or ".BIN file."

BIND
Prime's linker for producing Executable Program Formats (EPFs).

called program
A program that is invoked by another program and that acts as a subroutine.

calling program
A program that invokes another program or subroutine.

calling sequence
The sequence of arguments to be passed from a calling program to a called subroutine.

command environment
Information that pertains to the processing of commands and the execution of programs for a particular user. Also, rules that govern how many command levels a user may have. Limits on the command environment are usually created by the System Administrator of a site.

command environment breadth
The maximum number of simultaneous program invocations allowed per command level.

command environment depth
The maximum number of command levels allowed.

command function
A PRIMOS command that returns a text string, such as a CPL command function.

command level
: The area of memory addressed by a particular EPF. Because EPFs are assigned only to free areas of memory, several command levels may exist at once. Therefore, several programs may exist in one user's memory space at once. Each time a user suspends a program, the user process enters a new command level.

command level depth
: The number of command levels or suspended programs that are allowed.

command line
: A PRIMOS command with all its options, arguments, and command preprocessing features.

command line linking or loading
: Use of BIND with all necessary options on one command line.

command preprocessing features
: See wildcard, treewalking, iteration, and name generation position.

command processor
: A routine that processes arguments on a command line for a program. The command processor is CP$, discussed in Chapter 10, in the Subroutines Reference Guide, and in the Advanced Programmer's Guide, Volume III: Command Environment.

command stack
: The record of active invocations of commands and programs, showing their order and what invocation was last active.

command state
: Attributes of a command such as name, version, whether command preprocessing features are recognized, and so on. A complete listing of the command state structure is found in the Advanced Programmer's Guide, Volume III: Command Environment.

command-match
: A character string that is used as a pattern match for mini-commands to be listed with the LIST_MINI_COMMANDS command. The command-match string limits the response to commands matching the string (or part of the string, if wildcards are used).

common area or COMMON block
: A block of data that may be linked separately from a program. COMMON blocks are defined by the COMM pseudo-op in PMA, the COMMON statement in FORTRAN, the $E+ switch in Pascal, the EXTERNAL phrase in COBOL 74, the EXTERNAL attribute in PL/I-G, and the "extern" storage class in C.

DAM file
: A direct access file, faster for random access than SAM files.

default entrypoint search list
The entrypoint search list supplied by PRIMOS, named SYSTEM>ENTRY$.SR.

deferred common
A COMMON block that may be initialized later, and for which space is preallocated in a runfile. By default, COMMON blocks are deferred by BIND unless they are initialized.

DTAR
A Descriptor Table Address Register that designates whether its associated segments are unique (private) or shared (public). DTAR 0 and DTAR 1 reference shared segments (PRIMOS code and shared libraries and programs). DTAR 2 and DTAR 3 reference private segments (user space and impure user data and stacks).

dynamic entrypoint
A subroutine available for use by programs that do not themselves contain that subroutine. The name of a dynamic entrypoint is kept in a list of dynamic entrypoints. A program that wishes to call a subroutine that it does not itself contain invokes the PRIMOS dynamic linking mechanism, which searches the list of dynamic entrypoints to find the desired subroutine. The result of this search is the address of the desired subroutine in memory.

dynamic link
An address reference that is resolved at program runtime, whereas traditional subroutine calls are resolved at program link time. A dynamic link is stored as as faulted pointer to an entrypoint name.

dynamic linking
The passing of an address at program runtime rather than at program link time. Dynamic linking is the mechanism by which a runfile calls PRIMOS subroutines or entrypoints. Dynamic linking makes runfiles smaller and saves memory because shared routines can be accessed by many users. In addition, if a shared routine is altered, dynamic linking assures that the runfiles that use it need not be recompiled and relinked to use the new version of the shared routine.

dynamic runfile
See EPFs.

dynamic segment
A segment to be used for EPFs. Assignments of program data to dynamic segments are not known until runtime. Such segments must all exist simultaneously in the user's address space. See also static segment.

dynamic storage
Storage that is allocated as needed and released when not needed during program execution.

DYNT
(1) A pointer with the fault bit set, indicating reference to a dynamic entrypoint (same as a faulted pointer). (2) A subcommand of BIND that specifies a dynamic entrypoint.

entryname
The name of an entrypoint or label that may be invoked as a routine. This term may also be used to denote the final component of a pathname; it is rarely used that way in this guide.

entrypoint
A label in a program, a library, or PRIMOS itself. The entrypoint may be called by another program. Typically, an entrypoint identifies a traditional subroutine. It acts like a subroutine or called program, but is often part of a larger module, and is shared. Entrypoints may be created with ECB and SUBR in PMA; FUNCTION, PROGRAM, SUBROUTINE, or ENTRY in FORTRAN; PROGRAM-ID in COBOL; PROCEDURE or ENTRY in PL/I-G; PROGRAM, PROCEDURE, or FUNCTION in Pascal; and MAIN in C. Entrypoints may also be created in a library EPF with the ENTRYNAME subcommand of BIND.

entrypoint search list
A list of libraries to be searched for names called by programs. A user might create a search list to designate that the user's private library is to be searched before, after, or instead of, PRIMOS libraries for like-named routines. To create a search list, use SET_SEARCH_RULES as discussed in Chapter 6. See examples in Chapter 9.

entrypoints table
A table containing the names of all labels within a library EPF that can be called as independent routines. Use the ENTRYNAME subcommand of BIND to add labels to this table.

EPF
Executable Program Format. Also called a dynamic runfile. An EPF contains a description of a complete program or library and is assigned addresses at program runtime rather than at program link time. Thus, an EPF may execute in more than one address in memory, and two or more EPFs may exist in memory at once. EPFs are produced by BIND and are executed by the RESUME command.

EPF cache
A special list of recently used program EPFs, which are stored in memory in the hope that they will be called next. Retrieval of program EPFs from this area is faster than finding them again on disk.

EPF library
See library EPF.

error code
A standard code returned by PRIMOS to a program or subroutine to document success or failure of a program execution or the status of the system. Standard error codes are listed in the Subroutines Reference Guide. See also severity code.

expand an abbreviation
To determine the full command signaled by an abbreviation. See definition 2 of abbreviation.

external commands
Commands that are stored as runfiles in CMDNC0, and that may overlay other programs when invoked. These may be standard PRIMOS commands, or may be added by users.

faulted pointer
A pointer with the fault bit set on. Having the bit set on indicates that the address of the related symbol is not in the runfile. This condition may be a result of user error, or it may indicate that the symbol is a dynamic entrypoint and will be found at runtime.

file suffixes
Tags added to a filename, preceded by a period. Many suffixes cause BIND, PRIMOS, or another utility to process the file in a particular way. The most common suffixes are .BIN, .LIST, .RUN, .CPL, and the language suffixes.

file system object
A file, directory, segment directory, or access category.

file type
For an EPF, a file may be a program EPF, a program-class library EPF, or a process-class library EPF. For other file types, see the Subroutines Reference Guide.

file unit
A number assigned as a pseudonym to each open file by PRIMOS.

free segment pool
The total of segments allocated to a system. Each user must get his or her allocation of segments from this pool.

fullword
On the Prime 50 series, 32 bits.

function
A program that returns a text string as its value, to be substituted in place of the function invocation.

halfword
On the Prime 50 series, 16 bits.

I mode
An addressing mode with the same range as V mode (see below), but with more efficient use of decimal instructions. It is available for PL/I-G, Fortran 77, and Pascal.

image
The copy of a program or data before it is executed.

imaginary address
An address that BIND places in an EPF to identify a location that points to another location in the EPF. At program or library runtime, PRIMOS translates these imaginary addresses into actual memory addresses. An EPF can therefore execute in one segment at one time and in a different segment at another time.

imaginary segment number
See relative segment number.

inactive EPF
An EPF that is mapped to the user's address space, but that is not running or suspended. Such an EPF is frequently on the EPF cache.

interface subroutine
A subroutine that calls program 2 from within program 1. The subroutine that interfaces to the PRIMOS command environment for EPFs is CP$.

internal PRIMOS command
A command that is part of PRIMOS code (an entrypoint into PRIMOS). Internal commands are dynamic, which means that they do not overwrite programs in memory. A partial list of internal commands is in Chapter 10 of this guide. A complete list is in the PRIMOS Commands Reference Guide.

invocation
See program invocation.

iteration
The processing of a command line in such a way that lists in parentheses are processed one by one. Thus CBL (A B C) compiles files A.CBL, B.CBL, and C.CBL with the CBL compiler.

key
In subroutine calls, a symbolic name substituted for a code number. File keys have the format K$xxxx. Error codes have the format E$xxxx. Keys are listed in the Subroutines Reference Guide.

library
A series of routines in one binary file. Standard PRIMOS libraries are stored in the UFD named LIB. The user may create libraries also.

library EPF
A series of routines in one binary file, either part of PRIMOS or created by the user with the LIBMODE subcommand of BIND. The creation process is discussed in Chapter 6. Unlike a program EPF, a library EPF cannot be RESUMEd. A library EPF is linked to by inclusion in a search list. See also program-class library and process-class library.

link frames
See linkage.

linkage
Data structures, data values, and pointers to external subroutines and common areas (the modifiable part of a program).

linker
A utility that resolves external references within a runfile, so that different pieces of code (such as a calling program and a subroutine) can find each other at runtime. Prime's linkers, BIND, SEG, and LOAD, are all linkers, in that they all resolve external references as they are given binary files. BIND, however, leaves all loading to be done dynamically at runtime by PRIMOS.

main entrypoint
The first entrypoint in a file, or the one defined as the main entrypoint with the MAIN subcommand of BIND.

map
(Noun) A listing of addresses of important features of a runfile, such as unresolved references, procedure, or COMMON blocks.

(Verb) To allocate to an address space. From the time an EPF is run or another program is linked to it, an EPF is considered to be mapped until it is removed.

memory manager
The part of PRIMOS that allocates and releases memory and keeps track of which parts of memory are free.

mini-command level
A command environment where only a limited set of commands, discussed in Chapter 7, are recognized. A user attains mini-command level by using up all available command levels.

mini commands
The commands available when a user has used up all available command levels. The mini commands are discussed in Chapter 7.

mode
The mode of an EPF may be active, not active, or not mapped. Active files may be suspended or in use. These terms are discussed in Chapter 9, under the LIST_EPF command.

multiple program invocations
More than one EPF residing in memory at once.

name generation position
A number indicating the position in the command line of an argument to be used for name generation. Name generation is the duplication of target names from all or part of source names. The equals sign (=) in the target name triggers this process. Thus, COPY YOURS>@@ MINE>== copies all files in YOURS to identically-named files in MINE.

nonsuspended EPF
An EPF that is mapped to a user's address space but is not active or suspended.

nonshared library
See unshared library.

notify
Send a signal to; activate.

null pointer
A pointer with no address in it (segment 7777, offset 0).

object file
See binary file.

page
A division of a segment, containing 1K halfwords, or 2048 bytes.

paging disk or paging partition
An intermediate disk area to which information from memory is transferred to make room for other information in memory.

paging space
Space on a paging disk.

pathname
The name of a file, which may include the MFD or disk name, a directory name, and one or more sub-directory names, as well as the filename, each separated by the symbol >.

per-program data
Data that is changed by a program, so that a fresh copy of the data must be set up for each invocation of the program.

personal or per-user library
See private library.

personal search list
A search list created by the user with SET_SEARCH_RULES, as opposed to the system default search list.

PRIMOS
Prime's operating system for the 50 series.

private address space
The per-user memory allocated to a user.

private library
A user library that may be shared or unshared.

private segment
A per-user segment.

procedure
A program, subprogram, or routine, or the part of a program that contains instructions.

procedure segment
A segment that contains only the instructions (unmodifiable part) of a program.

process-class library
A set of routines that does not act as a physical part of the runfile (as does a program-class library). Only one copy of the linkage area exists for each user process. See Chapter 6.

program
An executable file. A program may be an EPF, a static-mode runfile, or a CPL file.

program breadth
The number of programs active at a given command level.

program EPF
An executable program created with BIND.

program invocation
Calling one program from another program or from PRIMOS level.

program name
The internal name of the main entrypoint. Entrypoints may be created with ECB and SUBR in PMA; PROGRAM, SUBROUTINE, or ENTRY in FORTRAN; PROGRAM-ID in COBOL; PROCEDURE or ENTRY in PL/I-G; PROGRAM, PROCEDURE, or FUNCTION in Pascal; and MAIN in C. Entrypoints may also be created in a library EPF with the ENTRYNAME subcommand of BIND.

program termination
Use RETURN, PRTN, or STOP to terminate a program EPF properly. The PRIMOS handling of termination is discussed in Chapter 4.

program-class library
A set of routines in an EPF, each of which acts as if it were physically part of a the program that invokes it. A new copy of the linkage (data storage) area is created each time one of these routines is invoked by a different program. The advantages of a program-class library are discussed in Chapter 6.

prompt
A display signaling that the computer is ready for more user input. OK, or ER! are the default PRIMOS prompts. The BIND prompt is the colon (:).

public address space
The shared memory that any user may access.

pure code
A piece of code that is not modifiable within the program in which it occurs. Such a program or subroutine may call itself or call a program that calls the original caller. This is called "recursion." If program code is pure, several users may share it. The results generated by such a program are not disturbed by the effects of previous computations. The most common way to assure pure code is to separate procedure and data into different segments and declare the procedure to be unmodifiable, while data segments use a separate stack for each subroutine call. Then with each call the calling module gets a new copy of the data variables.

All binary code produced by the CC, CBL, COBOL, F77, PASCAL, PL1G, and VPRG compilers is pure. The FTN compiler produces pure code if it is used with the -64V option. The PMA assembler produces pure code if it is used in V mode or I mode, and if programming guidelines outlined in the Advanced Programmer's Guide, Volume III: Command Environment, are followed.

R mode
An addressing mode in which only physical memory is available to the user. Only PMA and FORTRAN can produce R-mode code, and only LOAD produces R-mode runfiles.

RBF
Recovery Based File. A file used only by Prime's DBMS subsystem.

read-only VMFA
Access to disk that allows only reading, not modification, of the information on disk. See Virtual Memory File Access.

recursive
A recursive program can call itself.

reentrant
See pure code.

reinitialize environment
Reinitializing the environment returns the user's environment to the state it was in immediately after the user first logged in. See Chapter 9 for the INITIALIZE_COMMAND_ENVIRONMENT command.

relative segment number
A segment number that is reassigned by BIND. All segments available to users are taken by BIND as relative numbers unless the SYMBOL subcommand is used. Also called an imaginary segment number.

remote EPF
One not residing on the CPU of the user who invokes it.

remove an EPF
To disconnect an EPF from a user's address space (not to delete the EPF). To do this properly, use REMOVE_EPF.

replace
The function of the COPY command that replaces an open EPF with a newer version of the same name, creating a newly suffixed copy of the previous version.

Reset Force Load (RFL)
A BIND subcommand to specify that only entrypoints currently called in a runfile need be linked from a library into that runfile. The advantage is a smaller runfile. See Set Force Load.

resource limits
The limits imposed on creation of EPFs. The number of EPFS that a user may invoke is limited by the number of command levels allowed, the maximum number of program invocations allowed, and the amount of memory (private and dynamic segments) allocated.

returned text string
A text string returned by a function as an argument.

runfile
An executable object file. The runfiles produced by BIND are called Executable Program Formats (EPFs).

SAM file
A Sequential-Access-Method file, slower to access randomly than a DAM file, but needing slightly less storage space and somewhat faster to access sequentially.

search list
A list showing where a set of items is to be found. See entrypoint search list.

search rules
See search list.

segment
A block of address space consisting of 131,072 bytes (64K halfwords).

segment pool
The number of free segments available to all users of a system. Because this number is less than the maximum number of users times the individual segment maximum, a user wishing to acquire another segment may have to wait until other users return segments to the segment pool.

segment range
A group of segments assigned to one purpose.

segment/offset
An address consisting of a segment number followed by an offset number. Offsets are numbered by 16-bit halfwords.

Set Force Load (SFL)
To specify that all entrypoints in a library must be linked, whether or not they are currently called in the runfile. The advantage is provision for future modules linked in the runfile, which may reference modules already encountered.

severity code
A code returned by a command or function. A value of 0 indicates successful execution. A value greater than zero indicates no successful execution. A value less than zero indicates successful execution, but warns that the results may be surprising. See also error code.

shared library
One for which the same copy is used by all users, instead of a separate copy being called up for each user. Shared libraries save memory space. They are separate files from the user programs that call them.

slave process
A special phantom running under user control on a remote system.

snapping a link
Resolution of a dynamic link at runtime; a faulted pointer (DYNT) is changed to the segment/offset address of the target entry point.

source file
A file that contains programming statements in the format recognized by the PMA assembler or by one of Prime's high-level language compilers.

source object
For COPY, the file system object to be copied.

stack history
A record of calls to various programs and subroutines that are still active, showing where execution was interrupted.

standard library
The system library PFTNLB that is linked by BIND's LIBRARY subcommand when it is used with no arguments.

standard PRIMOS error code
See error code.

static command line buffer
A buffer (storage space) in a program that accepts command line arguments, but is addressed by RDTK$$ because the program is not an EPF.

static runfile
A runfile that is assigned addresses at program link time rather than at program runtime. All static runfiles tend to use the same addresses so that one overwrites another. SEG and LOAD produce static runfiles. See also EPFs.

static segment
A segment to be used for non-EPF programs, the address of which is assigned when the program is loaded. See also dynamic segment.

static storage
Segments reserved for storage of non-dynamic programs. Within a program, storage that is allocated once and for all at the beginning of program execution.

static-mode library
One that is reinitialized each time it is invoked by a different program.

static-mode program
A program that is not an EPF. See static runfile.

storage
Memory area allocated (reserved).

subprogram
A called program or subroutine.

subsystem
A utility such as DBMS, COBOL, or BIND.

suffix numbering
The way that COPY keeps track of EPF files that it is replacing. Suffix numbering is discussed in Chapter 9.

suspended EPF
An EPF that was running and was stopped by a terminal quit (such as CONTROL-P) or by an error condition. A suspended EPF is still considered active and may be continued with START.

symbol name
The name of a data item, program, COMMON block, or entrypoint in a binary file or a runfile. Symbol names must be given addresses, either at program link time or at runtime, before a program can run.

system default search list
The list of pathnames that PRIMOS uses to look for library routines referenced by a user, unless the user supplies a personal search list. The system default list can be changed by the System Administrator.

system library
A library furnished with PRIMOS, such as VSRTLI or VFORMS.

systemwide segment limit
The total number of segments available to all users in the segment pool.

target object
For COPY, the file system object copy to be created or replaced.

target program
For CP$, the program being invoked.

tilde
The ~ sign. Tildes are used to continue lines in a CPL program and to prevent command processor features from being applied to a command line.

token
An argument in a command line, such as a filename, a number, or an option.

treewalking
The processing of a command line that contains a pathname with a wildcard (using @ or + characters) in it, such that more than one subdirectory is searched. Thus, the command DELETE HERE>@@>JUNK will be expanded to delete files named JUNK in all subdirectories of HERE.

type
See file type.

unresolved reference
A call to another program or COMMON block that does not exist in the runfile. The unresolved or missing reference may be an omission by the user or may be a subroutine that is a dynamic entrypoint and which will be resolved by PRIMOS at runtime.

unshared library
A library for which each user gets a complete copy in his or her own address space. An unshared library is part of the user's runfile.

user version
The version number of an EPF, specified with BIND's VERSION subcommand by the user who created the EPF.

V mode
An addressing mode for virtual (segmented) memory that uses 32-bit registers, allowing 512 megabytes of memory to be addressed. V mode allows use of virtual segments, thus extending available memory beyond the amount of physical memory available. V mode is the recommended mode for most Prime users.

varying character string
A string of variable size and memory allocation. See the Subroutines Reference Guide for how to define a varying character string in each Prime language.

virtual circuit
An open network connection.

virtual memory
Memory as seen by the user, which is larger than physical memory. PRIMOS moves (pages) segments of code into and out of physical memory automatically, so that the user can address a larger space than physical memory. This larger space is virtual memory.

Virtual Memory File Access (VMFA)
Direct access to a disk rather than access to an intermediate paging disk. VMFA can be faster than paging. PRIMOS uses VMFA only for EPF procedure code.

VMFA
See Virtual Memory File Access.

VMFA-read
A read operation directly from disk, bypassing the paging mechanism.

warning severity code
A number that indicates a program has encountered undefined conditions. See severity code. For PRIMOS subroutines, warnings are a subset of error codes. See error code.

wildcard character
The at sign (@), plus sign (+), and caret (^). These characters are used one or more times in a character string to indicate "all names with or without the preceding or following characters." Thus, DELETE MYPROG@@ means delete MYPROG.FTN, MYPROG.BIN, MYPROG.LIST, and so on. See the PRIMOS Commands Reference Guide and the Prime User's Guide for a complete discussion of wildcards.

INDEX

Index

A

A register, initial value for, 5-7, 5-12, 5-18

A/SYMBOL (SEG) subcommand, 5-14

Abbreviations at mini-command level, 7-4, 9-3

Aborting from BIND, 2-13, 8-15

Absolute addresses, equating symbol names to, 8-17

Access to private segments, listing, 9-25

ACCESS_VIOLATION$ condition, 5-7

Activating the system search list, 9-32

Active EPFs,
- definition of, 9-9
- listing, 9-11

Actual memory addresses, 1-8

Addressing modes, 1-2

ALLOCATE subcommand, 5-18, 8-4

Allocating storage for COMMON blocks, 8-4

Application libraries in EPFs, 2-11

Assembler, PMA, 1-3

ATTACH (SEG) subcommand, 5-14

ATTACH$ search list, 9-5a, 9-5b, 9-23, 9-33, 9-34

AUTOMATIC (SEG) subcommand, 5-14

B

B register, initial value for, 5-7, 5-12, 5-18

Base areas, mapping, 8-12

Basenames, 2-2

Batch jobs and search lists, 6-11

Binary files (See Object files)

BINARY$ search list, 9-23

BIND, (See also EPFs; Library EPFs)
- adding suffix to EPFs, 2-12
- basic subcommands, 2-8
- changing .RUN suffix, 7-7
- creating library EPFs, 6-7
- description of, 1-4
- designating main procedure, 8-11
- displaying help screen, 2-16
- filing EPFs, 2-14
- linking libraries, 8-9
- linking object files, 2-9, 8-9
- map files, 8-11
- nonrecognition of prefixes, 2-3
- problems, 7-8
- quitting from, 2-13, 8-15
- recognition of suffixes, 2-3
- relinking modules, 2-18, 2-19
- required access rights to use, 2-5
- Summary of linking subcommands for, 8-2
- using from command line, 2-6
- using interactively, 2-7

BIND subcommand,
- COMPRESS, 8-14

BIND subcommands,
- ALLOCATE, 5-18, 8-4
- CHANGE_SYMBOL_NAME, 5-19, 5-20, 8-4
- COMMENT, 8-5
- COMMON_WARNING, 5-19, 8-5
- COMPRESS, 8-5
- DYNT, 6-12, 8-6
- ENTRYNAME, 6-8, 8-6
- FILE, 2-12, 5-15, 5-18, 6-9, 8-7
- HELP, 2-16, 8-8
- INITIALIZE_DATA, 8-8
- ITERATION, 8-19
- LIBMODE, 6-7, 8-8
- LIBRARY, 2-9, 6-2, 8-9
- LOAD, 2-9, 8-9
- MAIN, 5-20, 8-11
- MAP, 2-15, 5-9, 5-10, 8-11
- NAMGENPOS, 8-19

BIND subcommands (continued)
- NO_COMMON_WARNING, 5-16, 8-15
- NO_GENERATION, 8-19
- NO_ITERATION, 8-19
- NO_TREEWALK, 8-20
- NO_WILDCARD, 8-20
- QUIT, 2-13, 5-15, 8-15
- RELOAD, 2-19, 5-18, 8-15
- RESOLVE_DEFERRED_COMMON, 5-10, 5-17, 8-16
- SEARCH_RULE_VERIFY, 8-16
- SYMBOL, 5-19, 8-17
- VERSION, 8-18
- WILDCARD, 8-20

Breadth of command environment, 4-4, 7-2

Building EPF templates, 5-20

C

C programs,
- EXIT routines for, 8-5
- library for, 2-11
- return statements in, 4-3

CAM files, 8-21

CBL programs,
- library for, 2-11
- return statements in, 4-3

CBLLIB library, 2-11

CCLIB library, 2-11

CE$BRD subroutine, 7-5

CHANGE_SYMBOL_NAME subcommand, 5-19, 5-20, 8-4

CL$GET subroutine, 4-3

CL$PIX subroutine, 4-3, 8-18

CLEANUP$ condition, 9-6, 9-7

CLOSE command, 7-5, 9-6

CMDL$A subroutine, 4-3, 8-18

CMDNCO directory, 1-7, 10-3

CMDSEG utility, 5-9

COBOL compiler, 1-3

COBOL programs (See Program)

COBOL programs, return statements in, 4-3

COMANL subroutine, 4-3, 10-5

COMI$$ subroutine, 10-19

COMINPUT command, 10-19

COMLV$ subroutine, 4-4, 5-5

Command environment,
- breadth of, 4-4, 7-2
- depth of, 7-2
- exceeding breadth, 7-5
- exceeding depth, 7-5
- listing information on, 9-21
- resetting, 9-6

Command flags, 10-15

Command functions, 10-2

Command input files, 10-19

Command level,
- determining current, 7-2
- listing available, 9-21
- releasing, 7-5

Command lines,
- EPF subcommands for, 8-3
- passing to CPL programs, 10-5
- passing to EPFs, 10-5
- passing to static-mode programs, 10-5
- reading from, 4-3
- using BIND from, 2-6

Command output files, 10-19

Command preprocessing with CP$, 10-17

Command Procedure Language programs (See CPL programs)

Command processing features of EPFs, listing, 9-14

Command processor, 10-3
- features in EPFs, 8-18
- stack (See Stack)

Command status, values for, 10-13

COMMAND$ search list, 9-5a, 9-5b, 9-23

Commands, (See also EPF-related commands)
- CLOSE, 7-5, 9-6
- COMINPUT, 10-19
- COMOUTPUT, 7-4, 10-19
- CONCAT, 5-11
- COPY, 7-7, 9-36
- creating EPFs as, 8-18
- DELETE, 9-29
- DUMP_STACK, 7-8
- EXPAND_SEARCH_RULES, 9-5a
- FIX_DISK, 10-20
- INITIALIZE_COMMAND_ENVIRONMENT, 6-11, 9-6, 9-7
- LIST_EPF, 5-13, 8-5, 8-11, 9-8
- LIST_LIBRARY_ENTRIES, 9-19
- LIST_LIMIT, 9-21
- LIST_LIMITS, 7-6
- LIST_MINI_COMMAND, 9-22
- LIST_SEARCH_RULES, 9-23
- LIST_SEGMENT, 9-25
- mini-command level, 9-22
- PM, 7-5
- PSD, 5-9
- RDY, 7-2, 10-6
- REENTER, 7-4
- RELEASE_LEVEL, 5-5, 7-4, 7-6, 9-6
- REMOVE_EPF, 7-6, 9-29, 9-36
- RESTOR, 5-12
- RESUME, 1-3, 1-7, 3-1
- SEG, 1-3, 5-6
- SET_SEARCH_RULES, 6-9, 6-10, 7-9, 9-32
- START, 1-9, 4-4, 5-5, 7-4
- VPSD, 5-9

COMMENT subcommand, 8-5

Comments in EPFs, 8-5

COMMON (SEG) subcommand, 5-15

Common areas,
  EPFs and, 8-11
  misdeclared, 5-6
  redefinition of, 5-17, 5-19, 8-16

COMMON blocks,
  allocating storage for, 8-4
  disabling warning messages for, 8-15
  mixing with linkage data, 5-16
  placement of, 5-15
  resolving pointers to, 8-16
  warning messages for, 8-5

COMMON_WARNING subcommand, 5-19, 8-5

COMO command, 7-4

COMO$$ subroutine, 10-19

COMOUTPUT command, 7-4, 10-19

Compilers,
  COBOL, 1-3
  FTN, 1-3
  recognition of suffixes, 2-3
  RPG, 1-3

COMPRESS subcommand, 8-14

Compressing runfiles, 8-5

CONCAT command, 5-11

CONTROL-P at mini-command level, 9-3

Conventions for filenames, 2-2

Converting R-mode programs to V-mode programs, 1-3

Converting static-mode programs to EPFs, 5-7

COPY command, 7-7, 9-36

CP$ subroutine, 10-3
  error codes, 10-19
  invoking commands, 10-12, 10-13
  invoking functions, 10-14
  invoking programs, 10-13
  preprocessing commands, 10-17

CPL command functions written as EPFs, 1-5

CPL programs,
  accepting command lines, 10-5
  calling and called by EPFs, 4-4
  command preprocessing, 10-18
  executing, 3-2
  returning text strings, 10-7

CSN subcommand, 8-4

D

D/ (SEG) prefix, 5-15

DAM files, 5-2, 8-21

Data areas, creating, 8-4

DBG debugger, 1-3, 1-4, 1-10, 5-12, 7-10

Debuggers,
  DBG, 1-3, 1-4, 1-10, 5-12, 7-10
  PSD, 1-3, 1-4
  VPSD, 5-12

Debugging,
  EPFs, 1-4, 1-10, 5-12, 7-10
  I-mode programs, 1-3
  V-mode programs, 1-3

Defining symbol names, 8-17

DELETE (SEG) subcommand, 5-10

DELETE command, 9-29

Deleting mapped EPFs, 9-29

Depth of command environment,
  7-2

Direct Access Method files (See
  DAM files)

DUMP_STACK command, 7-8

Dynamic linkage, 6-4

Dynamic links to user library
  routines, 8-6

Dynamic runfiles (See EPFs)

Dynamic segments (See Segments)

Dynamic-mode runfiles (See EPFs)

DYNT subcommand, 6-12, 8-6

E

E$BARG error code, 10-21

E$BNAM error code, 10-20

E$BVER error code, 10-21

E$CMND error code, 10-21

E$DIRE error code, 10-20

E$ECEB error code, 7-5

E$EOF error code, 10-20

E$FIUS error code, 10-20

E$FNTF error code, 10-20

E$ITRE error code, 10-20

E$NDAM error code, 10-21

E$NINF error code, 10-21

E$NMTS error code, 7-7

E$NMVS error code, 7-7

E$NRIT error code, 10-20

ED editor, 1-8, 1-9, 2-5

EDB editor, 5-15, 8-10

Editors,
  ED, 1-9, 2-5
  EDB, 5-15, 8-10
  EMACS, 2-5
  NSED, 7-9

EMACS editor, 1-8, 2-5

EN subcommand, 8-6

ENTRY$ search list, 9-23, 9-33,
  9-34

ENTRYNAME subcommand, 6-8, 8-6

Entrynames in library EPFs, 9-19

Entrypoint names,
  length of, 1-4
  of EPF libraries, 8-6

Entrypoint search lists (See
  Search lists)

Entrypoints,
  definition of, 6-1
  listing in library EPFs, 9-19
  to library EPFs, 1-7
  using map files to identify,
    8-12

EPF libraries (See Library EPFs)

EPF templates, building, 5-20

EPF$ subroutines, 5-12

EPF$RUN subroutine, 10-3

EPF-related commands,
  INITIALIZE_COMMAND_ENVIRONMENT,
    9-6
  LIST_EPF, 8-11, 9-8
  LIST_LIBRARY_ENTRIES, 9-19
  LIST_LIMITS, 9-21
  LIST_MINI_COMMAND, 9-22
  LIST_SEARCH_RULES, 9-23
  LIST_SEGMENT, 9-25
  REMOVE_EPF, 9-29

EPF-related commands (continued)
  SET_SEARCH_RULES, 9-32
  summary of, 9-2

EPFs, (See also BIND; Library EPFs)
  accepting command lines, 10-5
  adding of suffix by BIND, 2-12
  advantages over static runfiles, 1-7
  allocation of segments by PRIMOS, 7-2
  calling other programs, 4-4, 10-1
  changing .RUN suffix, 7-7
  command preprocessing, 10-18
  command processing features of, 9-14
  comments in, 8-5
  common areas misdeclared in, 5-6
  CPL command functions as, 1-5
  creating as commands, 8-18, 10-3
  creating with BIND, 2-5
  debugging, 1-4, 1-10, 5-12, 7-10
  defining symbol names, 8-17
  definition of library EPFs, 1-7
  definition of program EPFs, 1-7
  deleting mapped, 9-29
  description of, 1-4, 1-6, 1-7
  executing, 3-1
  filing, 2-12, 8-7
  inserting comments, 8-5
  inserting version stamp, 8-18
  interface with PRIMOS, 5-2
  iteration for, 8-19, 9-14
  language libraries, 2-10
  limits on called programs, 7-2
  limits on invocations, 10-10
  link frames misdeclared in, 5-6
  linking, 2-9
  linking language libraries, 8-9
  linking libraries, 6-2, 8-9
  linking programs to, 6-4
  linking system libraries, 2-9
  linking to libraries, 6-4
  linking user libraries, 2-10
  listing information on, 9-8

EPFs (continued)
  main procedure for, 8-11
  map files, 8-12
  name generation for, 8-19, 9-14
  nonactive, 9-9, 9-11, 9-29
  order of library linkage, 2-10
  paging disk space, 1-4
  placement of areas in, 8-11
  problems with in-use, 7-7
  procedure areas, 8-11
  procedure segments used, 9-12
  processing ROAM files, 8-21
  producing maps of, 8-11
  programming guidelines for, 4-2
  relinking, 8-15
  reloading modules, 2-19
  remote, 1-10
  removing from user's address space, 9-29
  REPLACE files, 9-37
  replacing, 9-36
  replacing procedures, 8-15
  returning severity codes, 10-6
  returning text strings, 10-7
  sharing among users, 1-10
  stack frames misdeclared in, 5-6
  stages in creating, 1-8
  statuses of, 9-9
  subcommands for, 8-3
  suspending, 5-3
  treewalking for, 8-20, 9-14
  types of, 9-9
  use as CPL command functions, 1-5
  use of library EPFs, 6-12
  wildcarding for, 8-20, 9-14

Error codes,
  returning, 4-4
  summary of, 10-19

ERRPR$ subroutine, 4-3

ERRSET subroutine, 4-3, 4-4

Executable Program Formats (See EPFs)

EXECUTE (SEG) subcommand, 5-15

Executing EPFs, 3-1

EXIT subroutine, 4-3, 5-5

EXPAND_SEARCH_RULES command,
  9-5a

External PRIMOS commands, 10-3

F

F/ (SEG) prefix, 5-15

F77 programs, return statements
  in, 4-3

FILE subcommand, 2-12, 5-15,
  5-18, 6-9, 8-7

File suffix conventions, 2-2

Filenames, conventions for, 2-2

Filing EPFs, 2-12, 8-7

FIX_DISK command, 10-20

Flags, mapping command processor,
  8-12

Force-linking procedure code,
  8-10

FORTRAN IFTNLB library, 5-15

FORTRAN PFTNLB library, 5-17

FRE$RA subroutine, 10-4, 10-17

Freeing storage memory, 10-17

FTN compiler, 1-3

FTN programs,
  impure code in, 4-4, 5-4
  main procedure in, 8-11
  return statements in, 4-3

Functions,
  command preprocessing, 10-19
  invocation, 10-11, 10-12
  returning text strings, 10-7

G

GETERR subroutine, 4-4

H

HELP subcommand, 2-16, 8-8

I

I-mode dynamic programs (See
  EPFs)

I-mode static programs (See
  Static-mode programs)

ICE command, 9-6

IDATA subcommand, 8-8

IFTNLB FORTRAN library, 5-15

IL (SEG) subcommand, 5-15

ILLEGAL_SEGNO$ condition, 5-7,
  7-6

Imaginary memory addresses, 1-8

In-use EPFs, 7-7, 9-9

INCLUDE$ search list, 9-23

INITIALIZE (SEG) subcommand,
  5-15

INITIALIZE_COMMAND_ENVIRONMENT
  command, 6-11, 7-5, 9-6, 9-7

INITIALIZE_DATA subcommand, 8-8

Initializing data segments, 8-8

Initializing registers, 5-18

Internal PRIMOS commands, 10-2

Invoking,
  commands with CP$, 10-12, 10-13
  functions, 10-11
  functions with CP$, 10-14
  internal commands with CP$, 10-12
  programs, 10-7
  programs with CP$, 10-13

ISC sessions, terminating, 9-6

Iteration in EPFs, 8-19, 9-14

ITERATION subcommand, 8-19

ITR subcommand, 8-19

K

K (Keys) register, defining value for, 5-7

K$NRTN key, 4-3

K$SRTN key, 4-4

Keywords, search rule, 9-33

L

Language libraries,
  for EPFs, 2-10
  linking, 8-9

LE command, 9-8

LI subcommand, 2-9, 8-9

LIB UFD, 2-9, 8-9

LIBMODE subcommand, 6-7, 8-8

Libraries,
  CBLLIB, 2-11
  CCLIB, 2-11
  IFTNLB, 5-15
  linking to EPFs, 8-9
  list of language, 2-11
  nonconvertible to EPFs, 5-6

Libraries (continued)
  nonshared, 6-3
  order of linkage to EPFs, 2-10
  PASLIB, 2-11
  PFTNLB, 5-17
  PL1GLB, 2-11
  setting COMMON blocks, 8-4
  static shared, 6-3
  using, 6-2
  VAPPLB, 2-11
  VKDALB, 2-11
  VRPGLB, 2-11
  VSRTLI, 2-11

LIBRARIES* directory, 6-4

LIBRARY (BIND) subcommand, 2-9, 6-2, 8-9

LIBRARY (SEG) subcommand, 5-15

Library EPFs, (See also BIND; EPFs)
  creating, 6-7, 8-8
  debugging, 1-4, 1-10, 5-12, 7-10
  defining entrypoint names, 8-6
  definition of, 1-7, 6-3, 6-5
  entrypoints, 1-7, 9-19
  guidelines for using, 6-9
  linkage between program and process classes, 6-6
  linking programs to, 6-4
  links to, 8-6
  listing, 9-8, 9-11
  process-class type, 6-6
  program-class type, 6-5
  removing from address space, 9-29
  specific entrynames, 9-20
  use in program EPFs, 6-12

Library routines, creating links to, 8-6

Limits on called programs, 7-2

Limits on program invocation, 10-10

Link frames, misdeclared, 5-6

Linkage areas in EPFs,
listing, 9-12
mapping, 8-11

Linkage sequences, rewriting, 5-8

LINKAGE_ERROR$ condition, 7-9

Linkers,
definition of, 1-1
differences from loaders, 1-2

Linking,
EPFs, (See also EPFs)
EPFs to libraries, 6-4
libraries, 8-9
object files, 2-9, 8-2, 8-9
user library routines, 8-6

Links to user library routines, 8-6

LIST_EPF command, 5-13, 7-4, 8-5, 8-11, 9-8

LIST_LIBRARY_ENTRIES command, 9-19

LIST_LIMITS command, 7-4, 7-6, 9-21

LIST_MINI_COMMANDS command, 7-4, 9-22

LIST_SEARCH_RULES command, 9-23

LIST_SEGMENT command, 7-4, 9-25

Listing entrypoints in library EPFs, 9-19

Listing information on EPFs, 9-8

LL command, 9-21

LLENT command, 9-19

LM subcommand, 8-8

LMC command, 9-22

LO subcommand, 2-9, 8-9

LOAD (BIND) subcommand, 2-9, 8-9

LOAD (SEG) subcommand, 5-16

LOAD utility, 1-3

Loaders, differences from linkers, 1-2

Login file, 6-11, 7-9

LS command, 9-25

LSR command, 9-23

M

Main procedure of EPFs, designating, 8-11

MAIN subcommand, 5-20, 8-11

MAP (BIND) subcommand, 2-15, 5-9, 5-10, 8-11

MAP (SEG) subcommand, 5-10

Map files,
EPF, 8-12
identifying library EPF entrypoints with, 8-12
producing retroactive, 8-13

Map files, SEG, 5-11

Mapped EPFs, removing, 9-29

Memory addresses, 1-8

Memory, misdeclared, 5-6

MIDASPLUS files, 2-11

Mini-command level,
available commands, 7-3, 9-22
disabling of user abbreviations, 7-4, 9-3
leaving, 9-6
reaching, 5-5, 9-3
use of CONTROL-P, 9-3

Mini-commands, 7-3, 9-3, 9-5, 9-22

Misdeclared command areas in programs, 5-6

Misdeclared link frames in programs, 5-6

Misdeclared stack frames in programs, 5-6

MIX (SEG) subcommand, 5-9, 5-16

MODIFY (SEG) subcommand, 5-11

Multiple commands, preprocessing, 10-18

MV (SEG) subcommand, 5-16

N

Name generation,
  in EPFs, 8-19
  position for EPFs, 9-14

NAMGENPOS subcommand, 8-19

Naming conventions for files, 2-2

NCW subcommand, 8-15

NG subcommand, 8-19

NGP subcommand, 8-19

NITR subcommand, 8-19

NO_COMMON_WARNING subcommand, 5-16, 8-15

NO_GENERATION subcommand, 8-19

NO_ITERATION subcommand, 8-19

NO_TREEWALK subcommand, 8-20

NO_WILDCARD subcommand, 8-20

Nonactive EPFs,
  definition of, 9-9
  listing, 9-11
  removing from address space, 9-29

Nonshared libraries, 6-3

NPX slave processes, 9-6

NSCW (SEG) subcommand, 5-16

NSED editor, 7-9

NTW subcommand, 8-20

NW$ in names, 10-18

NWC subcommand, 8-20

NX$ in names, 10-18

O

Object files,
  creating, 1-8
  linking to EPFs, 8-9

Open EPFs, replacing, 9-36

OPERATOR (SEG) subcommand, 5-17

OUT_OF_BOUNDS$ condition, 5-7

Overwriting the system search list, 9-32

P

P/ (SEG) prefix, 5-17

Page boundaries, 5-17, 8-10

Paging disk space, EPF use of, 1-4

PARAMS (SEG) subcommand, 5-12

Pascal programs,
  library for, 2-11
  return statements in, 4-3

PASLIB library, 2-11

Pathnames, using BIND to display loaded object, 8-16

Pause statements in programs, 4-3

PBECB option, 4-4, 5-4

PFTNLB FORTRAN library, 5-17

PFTNLB library, 2-9, 8-9

Phantoms and search lists, 6-11

PL (SEG) subcommand, 5-17

PL1 programs,
  naming conventions in, 2-3
  restarting, 4-3
  return statements in, 4-3

PL1G programs,
  library for, 2-11

PL1GLB library, 2-11

PM command, 7-5

PMA assembler, 1-3

PMA programs,
  converting to EPFs, 5-3
  impure code in, 5-4
  main procedure in, 8-11
  return statements in, 4-3
  use of LINK and PROC pseudo-ops, 4-2

POINTER_FAULT$ condition, 5-7

Pointers, resolving to COMMON blocks, 8-16

Prefixes and BIND, 2-3

PRERR subroutine, 4-4

Prime Macro Assembler programs (See PMA programs)

Prime Symbolic Debugger (See PSD)

PRIMIX child processes, terminating, 9-6

PRIMOS commands (See Commands)

PRIMOS, interface with EPFs, 5-2

Private dynamic segments, 9-21

Private segments in use, 9-25

Private static segments, 9-21

Procedure areas in EPFs, 8-11

Procedure segments used by EPFs, 9-12

Process, command, 10-3

Process-class library EPFs (See Library EPFs)

Program EPFs (See EPFs)

Program invocation, 10-7
  command interface, 10-10
  limits on, 10-10
  recursive, 10-21
  use of shared memory, 10-10

Program-class library EPFs (See Library EPFs)

Programs (See C programs; CBL programs; CPL programs; EPFs; F77 programs; FTN programs; Library EPFs; Pascal programs; PL1G programs; PMA programs; Static-mode programs; VRPG programs)

PSD,
  command, 5-6
  debugger with R-mode programs, 1-3

PSD (SEG) subcommand, 5-12

Q

Q subcommand, 2-13, 8-15

QUIT (BIND) subcommand, 2-13, 5-15, 8-15

QUIT (SEG) subcommand, 2-14, 5-12, 5-18

Quitting from BIND, 8-15

R

R-mode programs (See Static-mode programs)

R-mode subroutines, 5-7

R/SYMBOL (SEG) subcommand, 5-18

Ranges for segments, mapping, 8-12

RD$CED subroutine, 7-5

RDC subcommand, 8-16

RDTK$$ subroutine, 4-3, 8-18, 10-5

RDY command, 7-2, 10-6

Recursive procedure invocation, 10-21

Recursive program invocation, 10-21

REENTER command, 7-4

Registers,
  setting values for, 5-7, 5-18
  setting values of, 5-12

RELEASE_LEVEL command, 5-5, 7-4, 7-6, 9-6

Relinking EPFs, 8-15

RELOAD subcommand, 2-19, 5-18, 8-15

Reloading modules, 2-19

REMEPF command, 9-29

Remote EPFs, 1-10

REMOVE_EPF command, 7-4, 7-6, 9-29, 9-36

Removing EPFs from user's address space, 9-29

REPLACE files, 9-37

Replacing open EPFs, 9-36

Resetting command environment, 9-6

Resetting system entrypoint search list, 9-32

RESOLVE_DEFERRED_COMMON subcommand, 5-10, 5-17, 8-16

Resolving pointers to COMMON blocks, 8-16

Resources,
  exceeding system, 7-7
  limit of user, 7-1

Restarting programs, 7-5

RESTOR command, 5-12

RESTORE (SEG) subcommand, 5-12

Restoring EPFs with DBG debugger, 5-12

RESUME (SEG) subcommand, 5-12

RESUME command, 1-3, 1-7, 3-1

RESUMEable SEG runfiles, 5-13

Retroactive map files, 8-13

Retrying link sequences, 2-18

RETURN (SEG) subcommand, 5-18

Return statements in programs, 4-3

Returning severity codes, 10-6

Returning text strings with EPFs, 10-7

Rewriting linkage sequences, 5-8

RFL flag, 8-10

RL (BIND) subcommand, 8-15

RL (SEG) subcommand, 5-18

ROAM files, 8-21

Routines,
  referencing, 8-4
  unresolved references, 2-15

RPG compiler, 1-3

RPG II (V-mode) programs, library for, 2-11

RPn files, 9-37

RUN suffix for filenames, 2-2

Runfiles,
  compression of, 8-5
  contents of, 1-1
  definition of, 1-1
  dynamic-mode (See EPFs)
  types of, 1-6

S

S-mode subroutines, 5-7

S/ (SEG) prefix, 5-18

SAM files, 8-21

SAVE (SEG) subcommand, 5-12, 5-18

Saving EPFs, 2-12

SCW (SEG) subcommand, 5-19

Search lists,
  ATTACH$, 9-5a, 9-5b, 9-23, 9-33, 9-34
  batch jobs and, 6-11
  BINARY$, 9-23
  COMMAND$, 9-5a, 9-5b, 9-23
  creating, 6-9
  designating user default, 6-10
  designating user library EPFs, 6-10
  ENTRY$, 9-23, 9-33, 9-34
  excluding system search list, 9-32
  getting fully-qualified pathnames from, 9-5a
  INCLUDE$, 9-23
  list of default, 9-23
  listing contents of, 9-23
  listing library EPFs in, 9-19
  overwriting system, 9-32
  phantoms and, 6-11
  problems with, 7-9
  resetting system, 6-11
  resetting system default, 9-32
  setting user-defined, 9-32
  STATIC_MODE_LIBRARIES entry, 9-33
  SYSTEM entry, 9-33
  uses of, 9-34
  Using search rule keywords with, 9-33

Search rule keywords, 9-33

SEARCH_RULE_VERIFY subcommand, 8-16

SEG command, 1-3, 5-6

SEG commands,
  MAP, 5-10

SEG loader, description of, 1-3

SEG map files, 5-11

SEG runfiles,
  creating, 1-3
  RESUMEable, 5-13

SEG subcommands,
  A/SYMBOL, 5-14
  ATTACH, 5-14
  AUTOMATIC, 5-14

SEG subcommands (continued)
  COMMON, 5-15
  DELETE, 5-10
  EXECUTE, 5-15
  IL, 5-15
  INITIALIZE, 5-15
  LIBRARY, 5-15
  LOAD, 5-16
  MIX, 5-16
  MODIFY, 5-11
  MV, 5-16
  NSCW, 5-16
  OPERATOR, 5-17
  PARAMS, 5-12
  PL, 5-17
  PSD, 5-12
  QUIT, 2-14, 5-12, 5-18
  R/SYMBOL, 5-18
  RESTORE, 5-12
  RESUME, 5-12
  RETURN, 5-18
  RL, 5-18
  SAVE, 5-12, 5-18
  SCW, 5-19
  SETBASE, 5-19
  SHARE, 5-13
  SINGLE, 5-13
  SPLIT, 5-19
  SS, 5-19
  STACK, 5-19
  START, 5-11
  SYMBOL, 5-19
  SZ, 5-20
  TIME, 5-13
  VERSION, 5-13
  VLOAD, 1-5, 5-14

SEG>CMDSEG utility, 5-9

Segments,
  allocation by PRIMOS, 7-2
  exceeding dynamic, 7-5
  exceeding static, 7-6
  initializing data, 8-8
  listing in-use, 9-25
  private dynamic, 9-21
  private static, 9-21
  unavailability of, 7-7

Semicolon as command separator, 10-18

Set Force Load flag (See SFL flag)

SET_SEARCH_RULES command, 6-9, 6-10, 7-9, 9-32

SETBASE (SEG) subcommand, 5-19

SETRC$ subroutines, 10-6

Severity codes,
  returning, 10-6
  signaling, 10-5
  values for, 10-6

SFL flag, 6-3, 8-6, 8-10

SHARE (SEG) subcommand, 5-13

Shared memory, use of, 10-10

SIGNL$ subroutine, 4-3

SINGLE (SEG) subcommand, 5-13

Sort libraries, 2-11

Source files, naming conventions, 2-2

Source Level Debugger (See DBG)

SPLIT (SEG) subcommand, 5-9, 5-19

Spooling EPF map files, 8-12

SRVFY subcommand, 8-16

SS (SEG) subcommand, 5-19

SSR command, 9-32

STACK (SEG) subcommand, 5-19

Stack frames, misdeclared, 5-6

Stack, use with EPFs, 5-3

START (SEG) subcommand, 5-11

START command, 1-9, 4-4, 5-5, 7-4

Static runfiles, 1-6

Static segments (See Segments)

Static shared libraries, 6-3

Static-mode libraries,
  errors while calling, 7-8
  linking to, 6-7

Static-mode programs,
  accepting command lines, 10-5
  addressing limitations, 1-2
  called by EPFs, 4-4
  calling EPFs and CPL programs, 4-4
  coexisting with EPFs, 1-9
  converting from R mode to V mode, 1-3
  converting to EPFs, 5-1, 5-8
  debugging, 1-3
  linked by LOAD, 1-3
  linked by SEG, 1-3
  purpose of R-mode programs, 1-2

STATIC_MODE_LIBRARIES entry in search lists, 9-33

Statuses of EPFs, 9-9

Stop statements in programs, 4-3

Subcommands, BIND (See BIND subcommands)

Subcommands, SEG (See SEG subcommands)

Subroutines,
  CE$BRD, 7-5
  CL$GET, 4-3
  CL$PIX, 4-3, 8-18
  CMDL$A, 4-3, 8-18
  COMANL, 4-3, 10-5
  COMI$$, 10-19
  COMLV$, 4-4, 5-5
  COMO$$, 10-19
  CP$, 10-3, 10-12
  E$ECEB, 7-5
  EPF$, 5-12
  EPF$RUN, 10-3
  ERRPR$, 4-3
  ERRSET, 4-3, 4-4
  EXIT, 4-3, 5-5
  FRE$RA, 10-4, 10-17
  GETERR, 4-4
  length of names, 1-4

Subroutines (continued)
  PRERR, 4-4
  R-mode, 5-7
  RD$CED, 7-5
  RDTK$$, 4-3, 8-18, 10-5
  S-mode, 5-7
  SETRC$, 10-6
  SIGNL$, 4-3

Suffixes in filenames, 2-2

Suspending programs, 7-2

SY subcommand, 8-17

SYMBOL (BIND) subcommand, 5-19, 8-17

SYMBOL (SEG) subcommand, 5-19

Symbol names, changing, 8-4

Synchronizers, deleting, 9-6

SYSTEM entry in search lists, 6-9, 9-33

System entrypoint search list, resetting, 9-32

System libraries in EPFs, 2-9

System resources, exceeding, 7-7

SZ (SEG) subcommand, 5-20

T

Templates, building EPF, 5-20

Text strings, returning, 10-7

TIME (SEG) subcommand, 5-13

Timers, deleting, 9-6

TREEWALK subcommand, 8-20

Treewalking in EPFs, 8-20, 9-14

TW subcommand, 8-20

Types of EPFs, 9-9

U

UFD CMDNC0, 1-7

Undefined symbols, mapping, 8-12

Unresolved routine references, 2-15

User abbreviations at mini-command level, 7-4, 9-3

User libraries, linking to EPFs, 2-10

Using libraries, 6-2

V

V-mode dynamic programs (See EPFs)

V-mode static programs (See Static-mode programs)

VAPPLB library, 2-11

VERSION (BIND) subcommand, 8-18

VERSION (SEG) subcommand, 5-13

Version stamp, inserting, 8-18

Virtual Memory File Access (See VMFA)

VKDALB library, 2-11

VLOAD (SEG) subcommand, 1-5, 5-14

VMFA segments, exceeding, 7-7

VPSD,
command, 5-9
debugger, 5-12

VRPG programs, library for, 2-11

VRPGLB library, 2-11

VSRTLI library, 2-11

W

WC subcommand, 8-20

WILDCARD subcommand, 8-20

Wildcarding in EPFs, 8-20, 9-14

X

X register, initial value for, 5-7, 5-12, 5-18

Z

Zero sector, 5-20

SURVEY

READER RESPONSE FORM

DOC8691-1LA Programmer's Guide to BIND and EPFs

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? ______________________

____

____

____

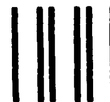
Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name: ______________________ Position: ______________

Company: ______________________________________

Address: ______________________________________

______________________________Zip: ___________



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