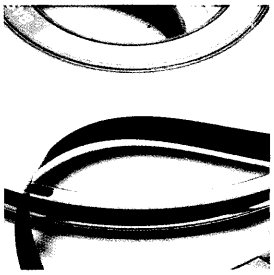
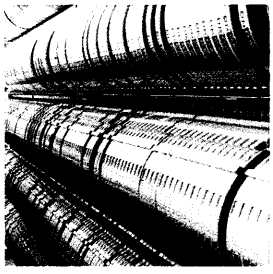
Prime Computer, Inc.

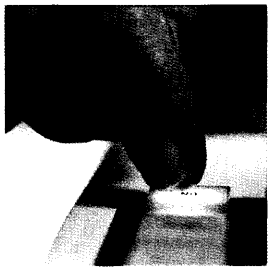
DOC10055-1LA
Advanced Programmer's Guide
Volume I
BIND and EPFs
Revision 19.4

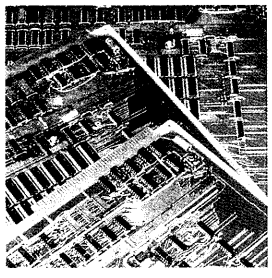













Advanced Programmer's Guide Volume I BIND and EPFs

First Edition

James Craig Burley

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.4.2 (Rev. 19.4.2).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

COPYRIGHT INFORMATION

CREDITS

Project Support	Alice Landy Len Bruns Margaret Taft
Editorial Support	Mary Callaghan
Graphic Support	Marjorie Clark Mike Moyle Bob Stuart
Production Support	Michelle Hoyt

PRINTING HISTORY — Advanced Programmer's Guide, Volume I:
BIND and EPFs

Edition	Date	Number	Software Release
Preliminary	January, 1985	DOC9229-1LA	19.4.0
First	September, 1985	DOC10055-1LA	19.4.2

In document numbers, L indicates loose-leaf.

CUSTOMER SUPPORT CENTER

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

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

HOW TO ORDER TECHNICAL DOCUMENTS

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

United States Only	International
Call Prime Telemarketing, toll free, at 800-343-2533, Monday through Friday, 8:30 a.m. to 8:00 p.m. (EST)	Contact your local Prime subsidiary or distributor.

Contents

ABOUT THIS BOOK ix

Prime Documentation Conventions x

1 INTRODUCTION TO BIND AND EPFS

What is an EPF? 1-2
Why EPFs? 1-4
History of Linking Loaders Under PRIMOS 1-4
BIND, the New Linker 1-8

2 THE DYNAMIC LINKING MECHANISM

What Is the Dynamic Linking Mechanism? 2-1
What Is a Dynamic Link? 2-2
What Happens To a Dynamic Link? 2-2
How Does PRIMOS Snap the Link? 2-3
Sample Session 2-4
What If the Desired Subroutine Cannot Be Found? 2-6
How Does Dynamic Linking Relate to Common Blocks? 2-7

3 THE EPF MECHANISM

EPF Organization 3-2
Subroutine Organization 3-4
The Life of an EPF 3-5
How Multiple Invocations of an EPF are Handled 3-34
How Simultaneous Use of an EPF Is Handled 3-35
How Debugging of an EPF Is Handled 3-35
How Running a Remote EPF Is Handled 3-36

4 EPFS AND STATIC-MODE APPLICATIONS

Restriction on the Use of Static-mode Programs by EPFs 4-2
Restriction on the Use of Static-mode Libraries by EPFs 4-4
Static Information to Avoid in EPFs 4-7
Effect of EPFs on Existing Shared Applications 4-8

5 PROGRAM EPFS

What is a Program EPF? 5-1
Writing the Main Program of a Program EPF 5-4

6 LIBRARY EPFS

What is a Library EPF? 6-2
Steps in Building a Library EPF 6-4
Choosing the Right Type of Library EPF 6-14
How to Use DBG on a Library EPF 6-30
Entrypoint Search Lists 6-32
Examining Entrypoint Lists 6-38
The Library EPF Mechanism 6-39

7 CODING GUIDELINES FOR EPFS

Writing Modules in High-Level Languages For EPFs 7-1
Writing Modules in PMA for EPFs 7-2

8 SHARED DATA

How to Define a Shared COMMON Area 8-2
How to Update Shared Information Atomically 8-7

9 MAPS AND ADDRESSES

Imaginary Vs. Actual Addresses 9-2
Using the LIST_EPF Command 9-3
Using the LIST_SEGMENT Command 9-5
Using the BIND Map 9-5
Using VPSD 9-8
Using the DUMP_STACK Command 9-9
Using Expanded Listings 9-13

10 BINARY EDITORS

LIBEDB 10-1
EDB 10-2
Examples 10-7

APPENDIXES

A CONVERTING PROGRAMS THAT USE REGISTER SETTINGS

How the Static Mode Program Works A-2
How to Achieve This Functionality In an EPF A-3

INDEX X-1

About This Book

The Advanced Programmer's Guide is intended for programmers who are experienced with Prime 50 Series systems, have read the Prime User's Guide (DOC4130-4LA) and Programmer's Guide to BIND and EPFs (DOC8691-1LA), are familiar with the Subroutines Reference Guide (DOC3621-190) and its first update package (UPD3621-31A), are experienced in at least one high-level language supplied by Prime (preferably PL1/G or FTN), and have an understanding of the architecture of Prime systems as described in the Prime 50 Series Technical Summary (DOC6904-191) and in the System Architecture Guide.

This guide consists of four volumes, and describes:

- Executable Program Formats (EPFs) in Volume I of this series
- The PRIMOS File System in Volume II of this series
- The PRIMOS Command Environment in Volume III of this series
- New features for readers of this guide in Volume 0 of this series
- Standard error codes used by PRIMOS, along with their messages and meanings, in Volume 0 of this series

Volume 0 also contains information applicable to all of the other volumes, such as an explanation of the presentation of the subroutine calls, general coding guidelines, and the like.

Designed for systems-level programmers, this guide describes the lowest-level interfaces supported by PRIMOS and its utilities.

Higher-level interfaces not described in this guide include:

- Language-directed I/O
- The applications library (APPLIB)
- The sort packages (VSRTLI and MSORTS)
- Data management packages (such as MPLUSLB and PRISAMLIB)
- Other subroutine packages

All of these higher-level interfaces are described in other manuals, such as language reference manuals, and the Subroutines Reference Guide.

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

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

PRIME DOCUMENTATION CONVENTIONS

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

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

1

Introduction to BIND and EPFs

This volume introduces EPFs (Executable Program Formats) and BIND, the new utility that creates them. It describes how to create EPFs and covers, in detail, concepts that apply to all EPFs and information applicable to two specific types of EPFs, program EPFs and library EPFs.

Specifically, Volume I of this guide:

- Explains what an EPF is
- Compares EPFs to the previously available method of building programs under PRIMOS
- Explains the dynamic linking mechanism
- Describes the EPF mechanism
- Explains restrictions on the use of static-mode programs and on the use of static-mode libraries
- Lists information and subroutines involving static information that should not be used in EPFs
- Describes the effect EPFs may have on existing static-mode shared applications

It is important that you read all of Chapters 1 through 6 to understand how EPFs affect all aspects of programming on Prime systems from Rev. 19.4 on. Even if your installation does not intend to use EPFs, you should be aware of the effect EPFs may have on existing shared applications used at your installation; this topic is covered in Chapter 4.

Prior to Rev. 19.4, SEG and LOAD were the only two utilities provided by Prime that linked programs. As of Rev. 19.4, a new linking utility named BIND is provided that creates programs using a new program format, the EPF. Chapter 1 introduces you to BIND and EPFs and compares the programming environment provided by BIND and EPFs to the environments provided by the SEG and LOAD utilities.

Other chapters in Volume I explain BIND and EPF concepts in greater detail:

- Chapter 2 explains the dynamic linking mechanism, which allows a program to call a subroutine that is not linked in with the program.
- Chapter 3 explains the EPF mechanism in detail, including elements of the Prime 50 Series architecture that relate directly to EPFs.
- Chapter 4 explains the effects that the advent of EPFs have on existing static-mode applications, even in an installation that does not switch over to using EPFs.
- Chapters 5 and 6 describe the ways in which program EPFs and the two classes of library EPFs (program-class and process-class) are created, and the operational characteristics of each.
- The remaining chapters introduce some coding guidelines that you should adhere to when programming EPFs, the concept of shared common data blocks and how to define them, and the use of linkage maps and the binary editor.

WHAT IS AN EPF?

An EPF is an executable file system object. You, the programmer, generate an EPF using BIND. An EPF may be used by a user or by another program. A file containing an EPF has a suffix of either .RUN or .RPn, where n is a digit (0-9). The .RUN suffix indicates that the file contains the latest version of the EPF. The .RPn suffix, if present, indicates an older version of the EPF; an old version of an EPF is kept only if at least one user is still using the EPF when the new version is installed.

Types of EPFs

There are two types of EPFs:

- Program EPFs, which contain a program having one entrypoint and which are invoked by explicitly running the program
- Library EPFs, which contain subroutines having one or more entrypoints and which are invoked by another program implicitly by referencing an entrypoint within the library EPF

Program EPFs: A program EPF contains a main entrypoint and related subroutines that together constitute a single program. A program EPF is invoked explicitly by issuing the PRIMOS command RESUME, by calling a subroutine to invoke a program EPF, or by issuing a command that names a program EPF residing in UFD CMDNC0.

To a programmer, a program EPF is a file containing a program. To a user, a program EPF is either a PRIMOS command (if the EPF resides in UFD CMDNC0) or a program invoked via the RESUME command. To a program, a program EPF is a subroutine, having a standard calling sequence, that may be invoked by calling one of several PRIMOS subroutines.

Chapter 5 contains detailed information about program EPFs (as distinct from library EPFs).

Library EPFs: A library EPF contains many subroutines, some (or all) of which are entrypoints to that library EPF. A library EPF is not invoked explicitly as is a program EPF; instead, the pathname of the library EPF is placed in an entrypoint search list (ENTRY$.SR) by the System Administrator (for the default system-wide search list SYSTEM>ENTRY$.SR) or by a user (for a private search list).

The dynamic linking mechanism, described later in this chapter, connects a library EPF to any program or subroutine that calls an entrypoint inside the library EPF. When any program or library calls a subroutine that is not contained within the program, it makes the call through a faulted Indirect Pointer (IP), also known as a dynamic link. Upon recognizing the faulted IP, the dynamic linking mechanism in PRIMOS takes action. First, it searches its own list of internal entrypoints (internal to PRIMOS). Next, it scans the user's entrypoint search rules for library EPFs (or the special -STATIC_MODE_LIBRARIES object), looking for a library that contains the subroutine named by the faulted IP as an entrypoint.

If the faulted IP identifies, as its target, the name of one of the entrypoints in a library EPF, PRIMOS connects the library EPF to the program invoking the subroutine via the faulted IP, allowing the program to call any of the entrypoints in that library EPF. As further faulted IPs are encountered, PRIMOS converts those that identify subroutines in that library EPF to point to the actual memory addresses of Entry Control Blocks (ECBs) in that library EPF. (An ECB is the

actual target of a subroutine call instruction, as it contains information on the subroutine such as where it is located, how much stack space it needs, and where its linkage information is located.) Subsequent uses of the affected IPs do not cause invocation of the dynamic linking mechanism; hence, they execute much faster. In fact, subsequent uses of such converted IPs execute as fast as IPs that were not faulted in the first place, such as IPs to statically allocated storage or to storage within the same program containing the IPs.

To a programmer, a library EPF is a collection of related subroutines that are useful to more than one application. To a user, a library EPF is nothing more than an entry in the entrypoint search list, with which many users do not even concern themselves. To a program, a library EPF appears as a collection of entrypoints to which the program may link itself by calling them via dynamic links (faulted IPs). However, a program is not concerned with how entrypoints are distributed among library EPFs; the programmer who builds a library EPF must concern himself or herself with the optimal grouping of related entrypoints in one or more library EPFs.

Chapter 6 contains detailed information about library EPFs (as distinct from program EPFs).

WHY EPFS?

EPFs are provided as an alternative to static-mode programs, which, until Rev. 19.4, were the only kind of program supported by PRIMOS. Static-mode programs are created by the SEG and LOAD linking loaders, while EPFs are created only by the new (at Rev. 19.4) BIND linker.

This section explains the history behind static-mode programs and EPFs, explains the disadvantages of static-mode programs, and explains the advantages of EPFs. During this discussion, information is presented suggesting how both static-mode programs and EPFs work.

History of Linking Loaders Under PRIMOS

Prior to Rev. 19.4, PRIMOS provided two linking loaders:

- SEG, for linking and loading V-mode and I-mode programs
- LOAD, for linking and loading R-mode programs

These linking loaders are fully described in the SEG and LOAD Reference Guide.

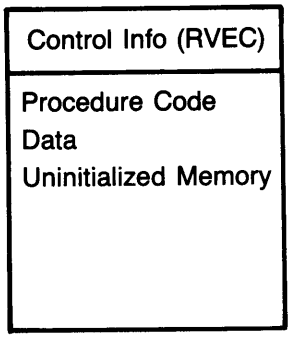
R-mode programs are limited to 128KB of memory in size. R mode is provided for compatibility so that programs written to run on the older Prime 100, 200, and 300 systems can run on newer Prime systems without modification. Such programs cannot take advantage of the large

segmented memory address space provided by PRIMOS starting with the Prime 400 system. (There are two submodes of R mode, 32R mode and 64R mode. They differ only in their ability to reference memory.)

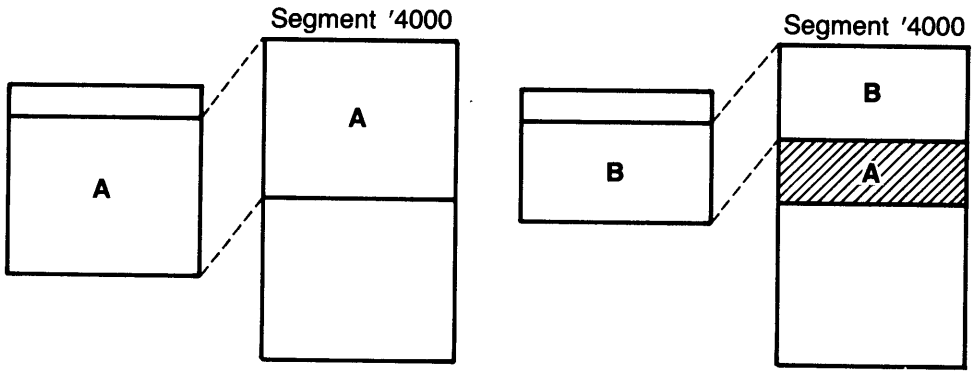
V-mode and I-mode programs can take full advantage of the large segmented memory address space provided by PRIMOS. V-mode and I-mode differ only in the way instructions are decoded and in the fact that registers in I mode are organized around a general-purpose register set architecture, while V-mode registers retain the special-purpose register set architecture inherited from the predecessor of V mode, which is R mode. (The predecessor of R mode is an almost entirely obsolete mode called S mode, which is used only during the very earliest phase of system boot and in certain system test and maintenance utilities. S mode consists of two submodes, 16S mode and 32S mode, which, like their counterparts in R mode, differ only in their ability to reference memory.)

R-Mode — Single-Segment Limit and DELSEG Requirement: Although SEG links and loads V-mode and I-mode programs, pre-Rev. 19.4 PRIMOS provided no direct way to execute V-mode or I-mode programs; only the R-mode (or S-mode) program format was supported. The R-mode program format produces a static-mode program. There are five characteristics of a static-mode program:

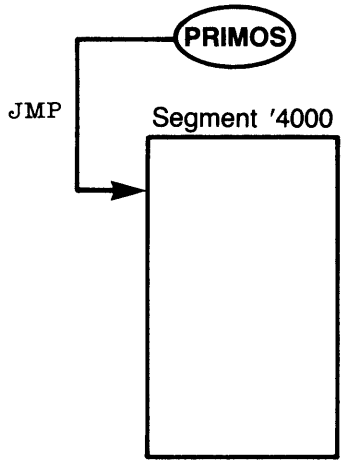
- It is represented in a SAM (Sequential Access Method) file that consists simply of a representation of the contents of a portion of segment '4000 when it contains the program. Nine halfwords of control information are followed by the memory image itself; the beginning and ending addresses of the memory image are in the nine halfwords of control information, as is the starting address of the program and the initial state of certain special R-mode registers. No distinction is made in the static-mode image between procedure code, data, and uninitialized memory.



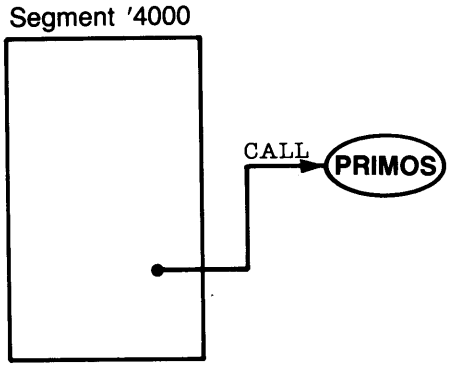
- It is always loaded into user segment '4000 by PRIMOS; therefore, if static-mode program A is first loaded, followed by static-mode program B, it is likely that loading program B will overwrite part or all of program A in memory unless they occupy completely separate areas in segment 4000. The default method of loading static-mode programs, however, starts all programs at location '1000 in segment '4000; therefore, static-mode programs loaded using the default method will invariably overwrite each other.



- It is executed by PRIMOS when the user issues the RESUME command identifying the static-mode program file, which typically has the suffix .SAVE to identify it as a static-mode program. PRIMOS loads the program into segment '4000 and performs a nonlocal goto to the starting address of the program; the program is not treated as a subroutine by PRIMOS.



- It terminates execution by calling one of several PRIMOS subroutines that return the user to PRIMOS command level, rather than by executing a return as a called subroutine might do.



- It can use more than one segment by directly referencing segments other than segment '4000. However, PRIMOS does not manage these additional segments. If a program uses five segments, then five segments remain allocated to the user when the program finishes.

  In order to free any segments referenced by an R-mode program (but no longer in use because the program has terminated), the user must issue appropriate DELSEG commands to delete the segments. (The DELSEG command is described in the PRIMOS Commands Reference Guide.) The additional segments are also released when the user logs out.

Due to the design of static-mode programs, one static-mode program cannot call another static-mode program as if it were a subroutine; the first static-mode program must give up control to the second program entirely, because the second program will destroy part or all of the first program.

SEG -- R-Mode to Initialize V-Mode or I-Mode: Because the static-mode mechanism does not handle V-mode or I-mode programs, the SEG loader is designed not only to provide the mechanism to link a V-mode or I-mode program, but also to enable the user to execute the program by typing:

```
SEG program-name
```

The SEG program itself is a static-mode program that loads program-name into memory (into segments other than segment '4000, which is where SEG resides). Then, SEG sets up the V-mode/I-mode environment and begins execution of the program.

- If a V-mode or I-mode program fits within a single segment, SEG can generate a static-mode image of the program in segment '4000 that can

then be invoked directly using the RESUME command. However, the procedure for building such a program via SEG is complicated and definitely not intuitive.

SEG for Shared Procedure Segments: If a V-mode or I-mode program is larger than a segment, but its impure data fits within a single segment, the program can be shared via SEG. SEG can generate, in segment '4000, a static-mode image of the impure portion of the program, which includes an interlude to the pure portion of the program; it can also generate static-mode images of the pure portions of the program in shared system-wide segments (for example, segments 2030, 2031, 2032, and so on). The procedure for building a shared program via SEG is extremely complex and has the following requirements:

- The System Administrator must coordinate the use of shared segments on the system and must assign shared segment numbers to programs that are to become shared programs.
- At every system coldstart, the shared static-mode segment images must be loaded into their corresponding shared segments.
- At system coldstart, the shared segments must be protected against modification.
- The user must RESUME the image of segment '4000 generated by SEG to run the program.
- To install a new version of the program reliably, the system must be shut down and restarted. Otherwise, the possibility exists that a user may be executing the old version of the program when the new version is installed. This usually results in unrecoverable errors for that user.

In addition, once a program is shared in system-wide segments, any user can examine or make copies of the pure code, even if that user cannot access the program itself (which is the impure and startup code residing in the image of segment '4000).

(See the SEG and LOAD Reference Guide for complete information on SEG and LOAD.)

BIND, the New Linker

As of Rev. 19.4, PRIMOS supports a new program type, called an EPF (for Executable Program Format). To build EPFs, PRIMOS provides a new linker named BIND. BIND is not a loader, because it does not load the final linked program into memory; PRIMOS is solely responsible for loading an EPF into memory.

BIND, like SEG, can be used to create only V-mode and I-mode programs, frequently referred to as 64V-mode or 32I-mode programs.

Using the BIND linker to create EPFs provides the following benefits:

- The BIND linker is much simpler to use than SEG, and is even simpler than LOAD (while providing more capabilities). This simplicity is maintained even when large programs are linked via BIND, because BIND and PRIMOS manage very small and very large programs in the same way.

- BIND allows external names (subroutine and common area names) to be a maximum of 32 characters in length; SEG limits external names to 8 characters, and LOAD to 6 characters, both by truncating external names.

- Two distinct types of EPFs are provided, one type to contain programs (invoked directly by a user) and another type to contain subroutine libraries (invoked implicitly by any program that references a subroutine in a library). With SEG, building a library is difficult, and its invocation must be explicitly performed by any program that wishes to invoke it by using an unusual program load sequence.

- Program EPFs are directly invoked using the RESUME command, rather than by an intermediate program (as is sometimes necessary with the SEG loader).

- A program EPF may be debugged by DBG without having to use a different build sequence than that used to build the production version of the same program EPF.

- Library EPFs are implicitly invoked when a program calls a subroutine in a library EPF; neither the library nor any subroutine in it needs to be made physically part of a program that uses the library.

- Any user may create his or her own personal library EPF and use that library EPF by placing its pathname in the user's entrypoint search list (a list of libraries to search for subroutine entrypoints).

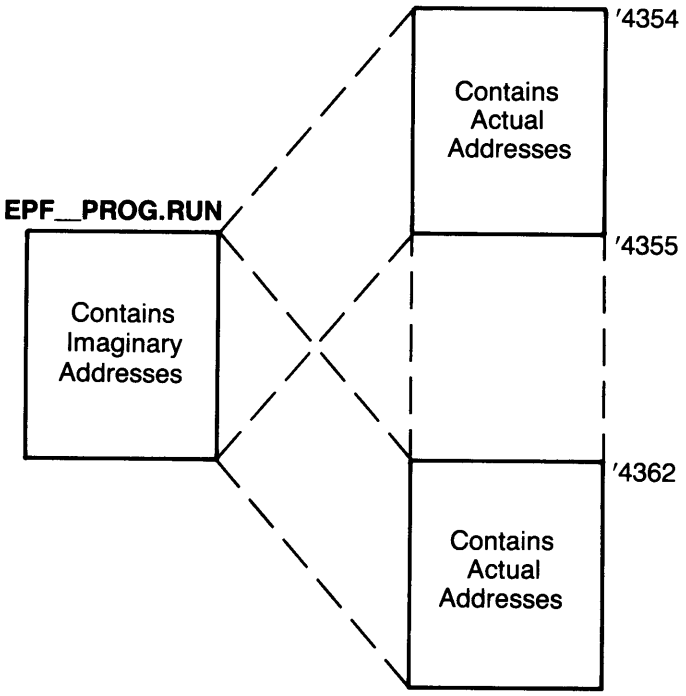
- A program EPF is invoked by PRIMOS as a subroutine, and may return to PRIMOS as a subroutine.

- PRIMOS separates memory used for EPFs, called dynamic memory, from memory used for static-mode programs, called static memory. Static memory for a user begins in segment '4000 and extends upward for the number of static segments allocated for the user by EDIT_PROFILE. Dynamic memory for a user begins at or beyond where static memory ends. Segments between the last static segment and the first dynamic segment for a user are not accessible.

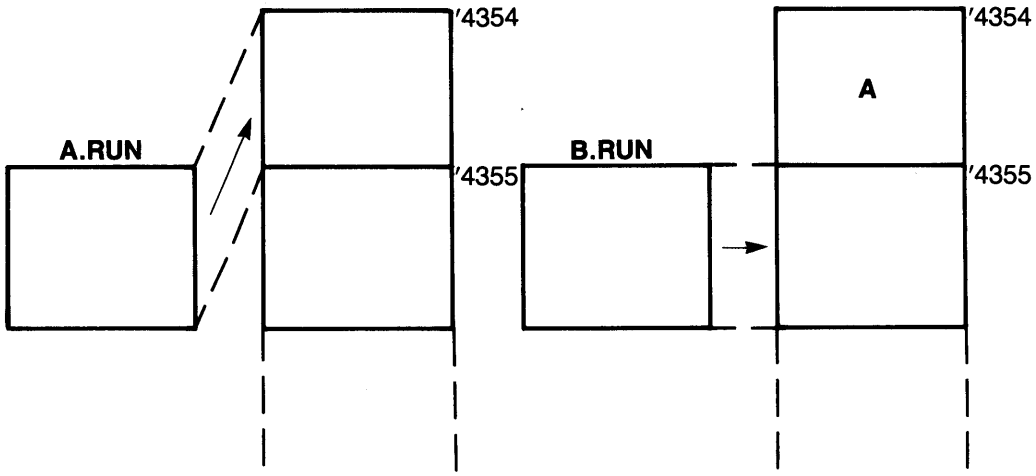
Static-mode programs cannot acquire additional dynamic memory in the same way they acquire additional static memory (by simply referencing it); therefore, EPFs are guaranteed not to be corrupted by loading static-mode programs.

While EPFs can use static memory, their use of static memory must be managed by the programmer; PRIMOS cannot keep track of which programs are using static memory as it can for dynamic memory.

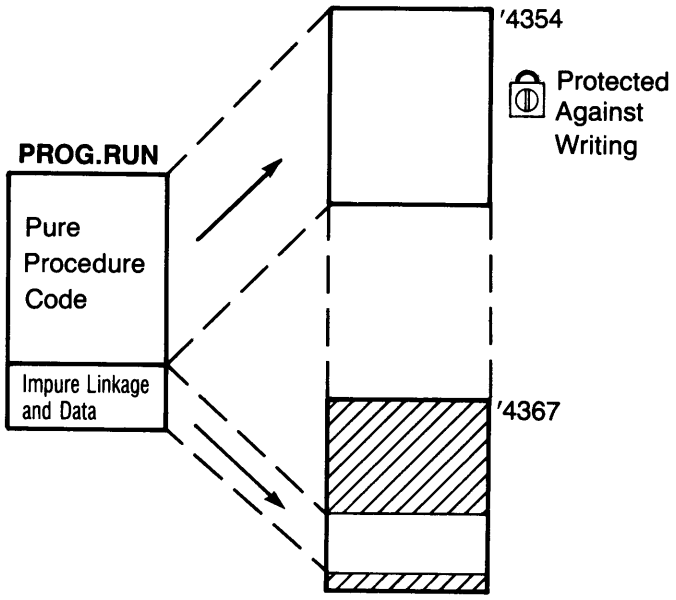
- Most memory allocation is handled entirely by BIND, at program or library linking time, or by PRIMOS, at program or library invocation time. Memory allocation performed by BIND is done entirely within segments, while PRIMOS performs the task of finding available segments for the program. So that this may work, BIND generally does not put actual memory addresses into an EPF; it uses imaginary addresses, which identify locations within the EPF. PRIMOS translates these imaginary addresses into actual addresses once it knows where, in memory, it will place the EPF. As a result, a particular EPF can execute properly in segment '4354 at one point in time, and can later execute properly in segment '4362, without needing to be relinked.



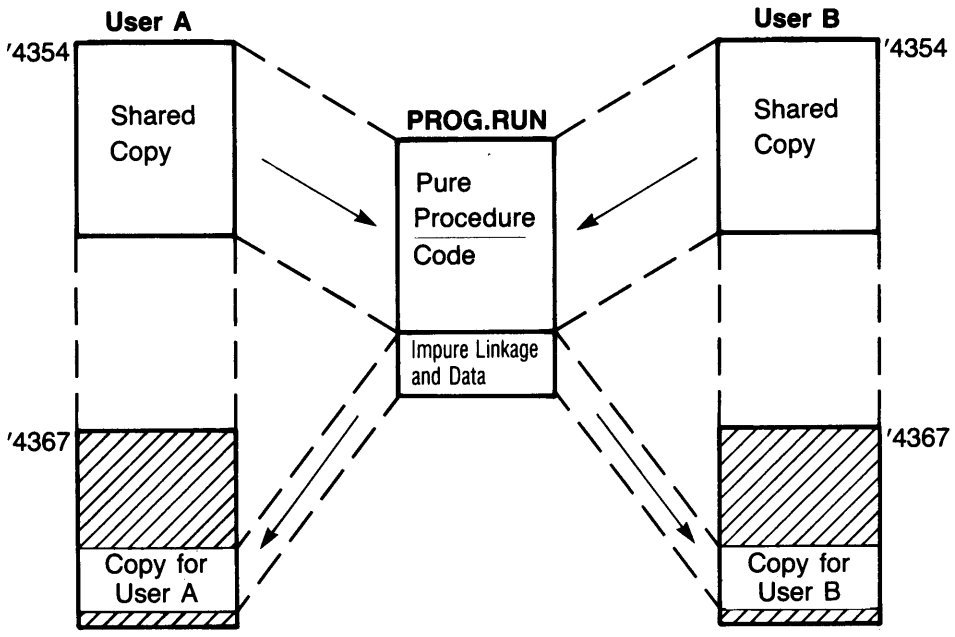
- Because memory allocation is handled by PRIMOS at runtime, PRIMOS ensures that EPFs do not overlay each other's memory space; this allows many EPFs to be kept in memory at one time without resulting in the destruction of data among the EPFs.



- An EPF separates procedure code that is pure (meaning it is not modified during program execution), from linkage text, impure code, and common areas, which are not pure. PRIMOS uses this separation to protect segments containing pure procedure code against modification by even the program itself, improving the chances of preventing a programming bug from turning into a disaster. A segment is used either to contain the procedure code of, at most, one EPF, or to contain the linkage text and common areas for many EPFs. A procedure segment may contain either pure or impure code, but only segments containing pure code are shared and protected against writing.

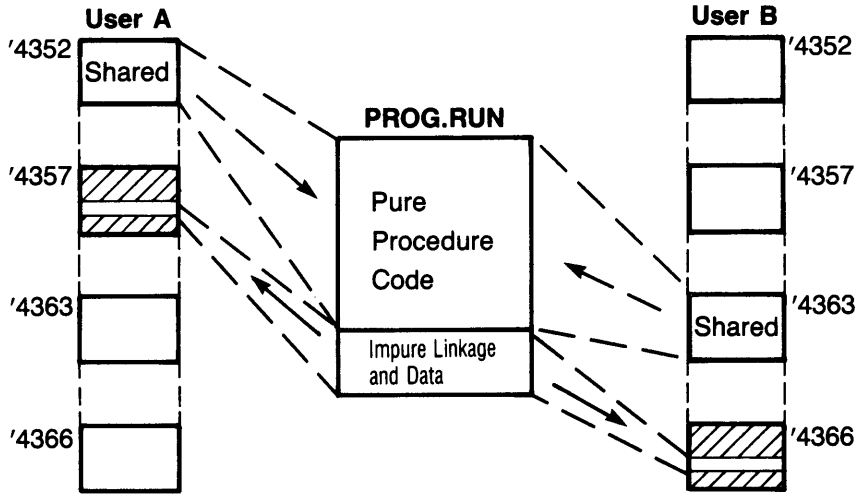


- PRIMOS also uses the separation of pure and impure code to automatically share pure code from a particular EPF between each user on a system that is using that EPF.

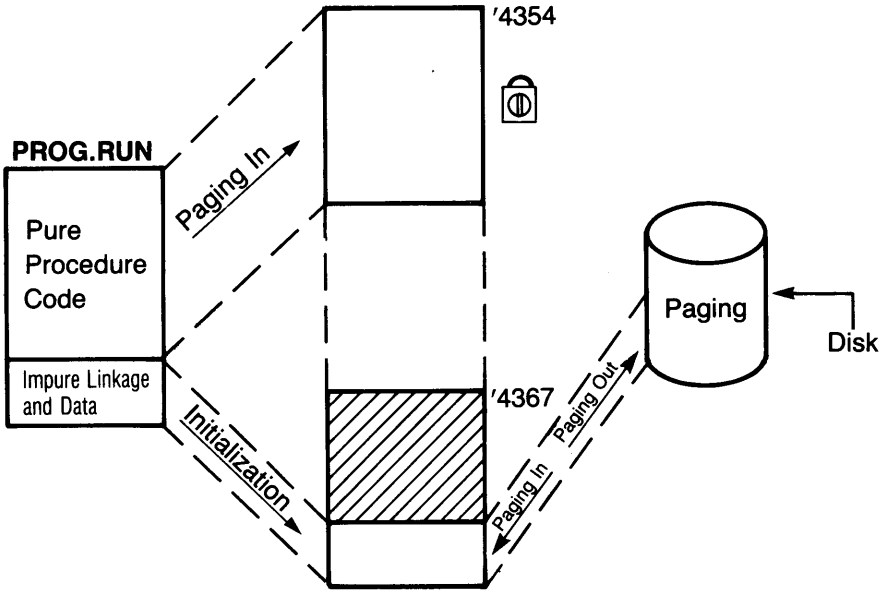


- PRIMOS shares the pure code of an EPF in the per-user (private) memory of each user who is using that particular EPF. Therefore, other users cannot access this shared code unless they, too, invoke the EPF (implying that they have sufficient access to do this).

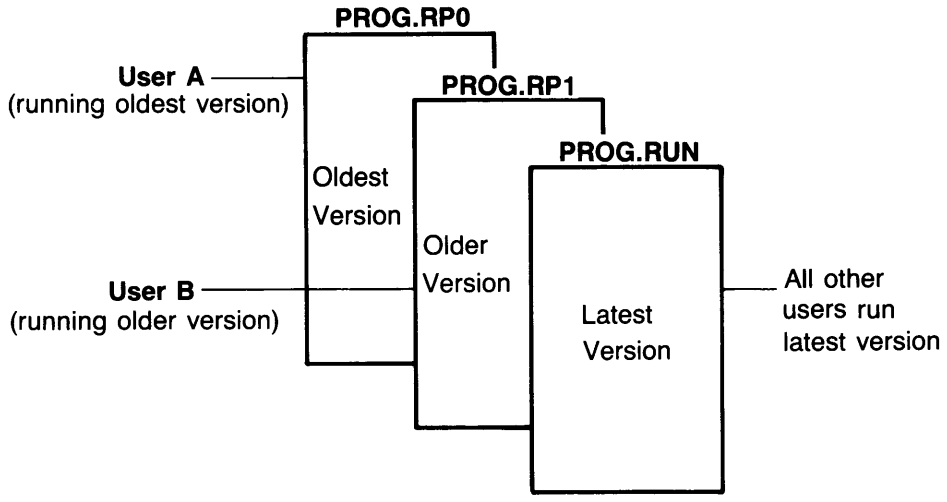
- Because BIND produces only imaginary addresses to point to locations within an EPF, PRIMOS, when it shares an EPF among two or more users, can use different segment numbers for each user while still sharing the pure procedure code. For example, an EPF might reside in segments '4352 and '4357 for user A, while also residing in segments '4363 and '4366 for user B. In this case, segment '4352 for user A and segment '4363 for user B could correspond to a single copy of the pure procedure code for that EPF; one copy of pure procedure code is therefore known by two different segment numbers for two users.



- PRIMOS further uses the separation of pure and impure code to avoid treating segments containing pure code as traditional virtual memory segments whose contents must be written out to the paging disk during paging. Instead, PRIMOS can always be sure of reading pure EPF segments from a single copy of the EPF, since pure segments are "locked" against modification, and hence always up-to-date. This improves performance and uses less paging disk space.



- Although PRIMOS automatically shares EPFs between users, new versions of EPFs can be installed at any time by using the COPY command to replace an old version. If the old version is still in use by at least one user, PRIMOS keeps the old version on the disk, renaming it so that subsequent invocations of the EPF invoke the new version. A maximum of 10 old versions of an EPF may be kept in this fashion. Although PRIMOS creates the old versions, it is up to the owner of the EPF to delete them when they are no longer in use.



- An EPF can be removed from memory by issuing the REMOVE_EPF command. All memory associated with the removed EPF that belongs to the user issuing the command is thereby returned to the free memory pool. If no other users are using the EPF, this means that one or more segments, which contained the procedure code for the EPF, are returned to the system-wide free segment pool.

- The command line arguments used to invoke a program EPF can be received by that EPF as an argument to the main entrypoint of the EPF, due to the symmetric call/return flow of program EPF invocation by the PRIMOS command environment. Such a program need not make a special subroutine call to acquire the command line from the command environment. This feature prevents the side effect of wiping out any unparsed data on the original command line by aborting a running program and then typing START; such side effects can occur with programs that continue to use RDTK$$ to retrieve the command line token by token.

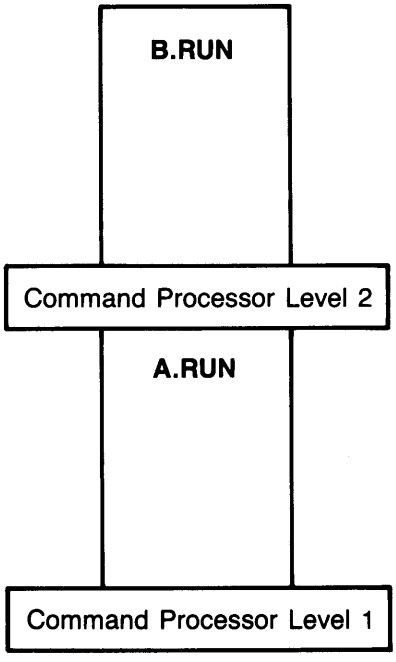
- Users can write their own CPL functions by building program EPFs that interface with the command processor.

- A program EPF can be executed by another program and treated as a command (which does not return a value) or as a CPL function (which returns a string value).

- A program EPF can be constructed to selectively enable or disable most forms of command-line preprocessing performed by PRIMOS, such as wildcards, treewalking, iteration lists, and name generation.

- A program EPF can obtain information on what kind of command preprocessing is taking place for the invocation of the EPF. For example, a program EPF can determine whether it is being invoked with a wildcard specification, even though what it receives as a command line argument is an objectname without wildcards; if wildcarding is being used, the program may wish to alter its output display to suit a list of file system objects.

- Program EPFs can be stacked on the command processor stack, allowing you to invoke program A, suspend it (via Control-P for example), invoke program B, and then, when program B finishes, use the START command to continue execution of program A at the point at which you suspended it. This same mechanism applies when program A is suspended as a result of a programming error (such as a memory access violation or use of an illegal segment number), allowing easier program debugging by using more advanced debugging techniques.



- PRIMOS maintains an EPF cache for each user. Terminated program EPFs are not immediately removed from memory, but instead are placed in the EPF cache. A subsequent invocation of a program EPF that is in the EPF cache results in faster startup time for that EPF, because most of the initialization of the EPF has already been completed. PRIMOS keeps the EPF cache from overloading system resources by removing older EPFs from the cache (and also from memory) as new EPFs are invoked.

- An EPF contains not only procedure code and linkage information, but general information on the EPF itself, including:

  - The version of BIND that was used to link the EPF

  - The date and time the EPF was linked via BIND

  - The name of the program, which may be different from the name of the file containing the program

  - The version of the program, as assigned by the programmer during the BIND session

  - A comment pertaining to the program, as assigned by the programmer during the BIND session; for example, a copyright notice

- Debugging of an EPF that is in production mode is easier; after an EPF is suspended, due to a user quit or some error, you can:

  - Use the LIST_EPF command to determine where in memory the EPF is located

  - Use the DUMP_STACK command to display the stack history of the EPF and the PRIMOS command environment

  - Use BIND, which itself is an EPF, either to display a map of the EPF at the terminal or to write one into a file by typing:

    ```
    BIND -LOAD EPF-filename -MAP [map-filename] -QUIT
    ```

    (With SEG and LOAD, you cannot generate a map of the program without destroying part or all of the in-memory copy of the suspended program.)

  - Use VPSD, which remains a static-mode program, to examine the EPF in memory. VPSD does not overwrite your EPF as it may static-mode programs; you no longer have to consider whether to use VPSD or VPSD16. Note, however, that you cannot place breakpoints in the procedure code of either a library EPF or of a program EPF that has been invoked via RESUME, because the pure procedure code is

protected against writing. (Use the VPSD subcommand of DBG if you wish to place breakpoints in the procedure code of an EPF with VPSD.)

- Use other commands, such as ED, SPOOL, and so on, without disturbing the in-memory copy of the EPF or the stack history of the EPF. After invoking static-mode programs, you must issue the RELEASE_LEVEL command (abbreviated RLS) once to prevent a subsequent START command from returning you to the static-mode program rather than your EPF. However, avoid issuing commands such as RELEASE_LEVEL -ALL and INITIALIZE_COMMAND_ENVIRONMENT (abbreviated ICE), as these delete stack history and, in the case of ICE, remove EPFs from memory.

- The detection of uninitialized variables is improved because, unlike SEG, BIND and PRIMOS do not initialize uninitialized static data to all zeroes for EPFs. While this may produce the undesirable effect of a working program failing when converting from using SEG to using BIND, it does significantly improve the chances of the programmer detecting cases of uninitialized variables when a program is built as an EPF. When such a program does not depend on default initialization to zeroes, it is more portable in that it can be ported to hosts whose own operating systems do not provide default initialization.

As you read on in this volume, particularly Chapters 2 through 6, you may discover additional benefits of using BIND to create EPFs. For information on features provided by EPFs when interacting with the PRIMOS command environment, see Volume III of this series.

2

The Dynamic Linking Mechanism

This chapter explains the concepts of the dynamic linking mechanism, and details its operation. Because this mechanism is used by all programs and libraries, all readers of this guide should understand how it operates. A thorough understanding of dynamic linking on Prime systems improves one's ability to solve product design and packaging problems while it simplifies the debugging of errant programs.

WHAT IS THE DYNAMIC LINKING MECHANISM?

The dynamic linking mechanism in PRIMOS allows a program to call subroutines that are not linked in with the program itself, but which are located in one of several libraries when the program is run. The advantages of placing subroutines in libraries that are connected by the dynamic linking mechanism, rather than binding them into each program that uses them, are:

- Less disk space is wasted because only one copy of the subroutine exists on a system, rather than having a copy in every program that uses the subroutine.
- Less memory is needed when two or more users are running programs that use the same subroutine, because the subroutine is automatically shared between users.
- New versions of the subroutine can be easily installed, as long as they are compatible with previous versions, without having to relink all programs that use the subroutine.

- Internal privileged PRIMOS subroutines, which execute in ring 0, can be accessed via the dynamic linking mechanism if they are entrypoints into PRIMOS; special-purpose supervisor call instructions need not be used.

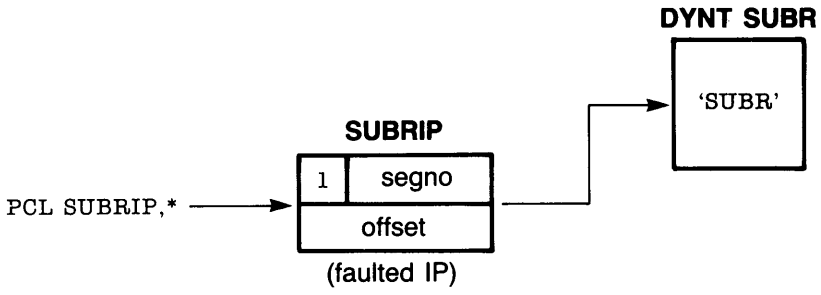
There are three general types of subroutine libraries in PRIMOS:

- PRIMOS-resident libraries, containing subroutines that are internal to PRIMOS.
- Library EPFs, supplied by Prime or by vendors, or user-written; these are described in Chapter 6.
- Static-mode shared libraries, supplied only by Prime.

The dynamic linking mechanism allows access to specific subroutines, called entrypoints, in these three types of libraries. Although a particular library may have several subroutines, a subset of these subroutines may be declared by the library as entrypoints; only entrypoints may be called by programs that reside outside the library itself.

WHAT IS A DYNAMIC LINK?

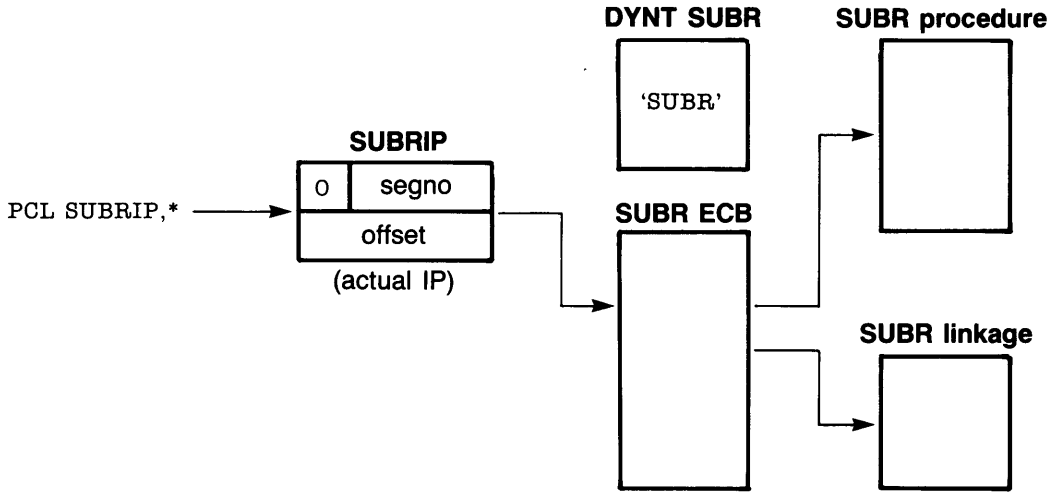
The crux of the dynamic linking mechanism is the dynamic link, also called a DYNT (pronounced "dint"). In place of a normal pointer to the ECB of a subroutine, a dynamic link consists of a special pointer, called a faulted IP (Indirect Pointer), that points to the name of an entrypoint. The dynamic link serves as a placeholder for a subroutine until that subroutine is located and connected to the program; it contains the name of the subroutine.



WHAT HAPPENS TO A DYNAMIC LINK?

When a program is run, it attempts to use faulted IPs when calling subroutines external to the program. Each time a faulted IP is encountered, a fault condition results; this causes PRIMOS to call its dynamic linking mechanism to resolve the fault condition.

The dynamic linking mechanism examines the faulted IP, determines that it is part of a dynamic link, and begins a complex process that culminates in the determination of the actual IP of the ECB of the target subroutine. PRIMOS then replaces the faulted IP with the actual IP; this is called snapping the link. At this point, PRIMOS resets the fault condition and continues the program. The instruction that caused the fault by referencing the faulted IP is executed again, and it succeeds: the target subroutine is called.



HOW DOES PRIMOS SNAP THE LINK?

To determine the name of the target subroutine, PRIMOS modifies the faulted IP so that it is a normal IP by resetting the Fault bit in the IP. (PRIMOS does this to a temporary copy of the IP, not the copy in memory that caused the fault.) Then, PRIMOS uses the resulting IP to read, from memory, the name of the target subroutine. (The name is stored in CHARACTER VARYING format.)

Once PRIMOS has the name of the target subroutine, it searches through its list of libraries (all three types) for a library that declares that name as an entrypoint. When PRIMOS finds the first such library, PRIMOS performs whatever initialization of that library is needed (such as initializing ECBs, IPs, static data, and so on). Then, PRIMOS determines the actual address of the ECB of the target subroutine within that library and replaces the faulted IP with that actual address.

SAMPLE SESSION

The following annotated sample session illustrates the effects of the dynamic linking mechanism. A program named LINKIT is written which first sleeps for ten seconds and then calls the T1OU subroutine. The ten-second sleep allows the user to type CONTROL-P before the T1OU subroutine is called; the user can then examine (via VPSD) the faulted IP that will be used to call T1OU, and can also examine the name of the target subroutine (the T1OU DYNT).

The user then terminates VPSD, types START to continue execution of LINKIT, and waits for the program to finish. After the program finishes, the user examines the IP again and sees that it is no longer faulted, but instead points into the linkage area for the library EPF named SYSTEM_LIBRARY, as shown by an ensuing LIST_EPF command. (The IP points into the linkage area rather than the procedure area because a resolved IP to a subroutine points to the ECB of that subroutine; the ECB itself actually points to the first executable instruction of the subroutine.)

```
OK, ED
INPUT
      SUBROUTINE LINKIT
      CALL SLEEP$(010000)   /* SLEEP FOR 10 SECONDS
      CALL T1OU(:207)   /* RING THE BELL
      RETURN
      END
(CR)
EDIT
FILE LINKIT.FTN
OK, FTN LINKIT -DYNM -DCL
0000 ERRORS [<LINKIT>FTN-REV19.3]
OK, BIND -LOAD LINKIT -LIBRARY
[BIND rev 19.4]
BIND COMPLETE
OK, INITIALIZE_COMMAND_ENVIRONMENT    (cleans up environment)
OK, RESUME LINKIT
(user types Control-P after a few seconds elapse)
QUIT.
OK, LIST_EPF -DETAIL     (shows placement of program in memory)

1 Program EPF.

(active)      <USRDSK>UNGER>LINKIT.RUN
  1 procedure segment:     +0:4340
  1 linkage area:          -2:4377(3)/70
  bind version:       19.4
  date of binding:    84-11-13.16:38:40.Tue
  program name:       LINKIT
  user version:       (none)
  comment :           (none)
  debug segments:     1
  command options:    wldcrd,trwlk,iter  file,dir,segdir,acat  1
```

```
OK, VPSD   (use the symbolic debugger to examine memory)

$SN 4340    (the procedure segment)

$A 1000:S   (EPFs typically start at offset '1000; see map)
4340/ 1000 PCL% LB%+ 422,*    (CR)    (the SLEEP$ call)
4340/ 1002 AP    1012,SL (CR)
4340/ 1004 PCL% LB%+ 424,*    (CR)    (the T1OU call)
4340/ 1006 AP   LB%+ 400,SL (CR)
4340/ 1010 PRTN  /
$LB 4377 70-400    (linkage references are offset by '400)

$A LB%+424:O    (access the faulted IP)
4377/ 114 104340 (CR)    (the fault bit is the 1 in 104340)
4377/ 115 1014 /
$SN 4340    (select the segment, without the fault bit)

$A 1014
4340/ 1014 4 :A    (name of DYNT is four characters long)
4340/ 1015 T1 (CR)  (name of DYNT is T1OU)
4340/ 1016 OU /
$Q
OK, RELEASE_LEVEL    (release the static-mode VPSD invocation)
Static mode program released. (rls)
OK, START    (continue the suspended invocation of LINKIT)
(bell rings)
OK, VPSD    (reenter VPSD)

$LB 4377 70-400    (set up LB again)

$A LB%+424:O    (access the same place in memory)
4377/ 114 4377 (CR)    (IP now points to 4377/16304)
4377/ 115 16304 /
$Q
OK, LIST_EPF -DETAIL -NO_WAIT    (check for library EPF)

1 Process-Class Library EPF.

(active)     <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
  2 procedure segments:   +0:4342              +2:4343
  2 linkage areas:        -2:4376(0)/0         -4:4377(3)/1134
  bind version:      19.4
  date of binding:   84-10-25.16:17:20.Thu
  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 EPF.
```

```
(not active) <USRDSK>UNGER>LINKIT.RUN
  1 procedure segment:     +0:4340
  1 linkage area:          -2:4377(3)/70
  bind version:       19.4
  date of binding:    84-11-13.16:38:40.Tue
  program name:       LINKIT
  user version:       (none)
  comment :           (none)
  debug segments:     1
  command options:    wldcrd,trwlk,iter  file,dir,segdir,acat  1
```

OK,

WHAT IF THE DESIRED SUBROUTINE CANNOT BE FOUND?

Dynamic links, also referred to as faulted IPs, are resolved as the program runs. They are resolved to point to one of the three types of libraries discussed previously. If none of the libraries known to PRIMOS lists a particular subroutine as an entrypoint, PRIMOS signals the error condition LINKAGE_FAULT$. Unless intercepted by a program, the condition results in a display similar to the following:

```
Error: condition "LINKAGE_FAULT$" raised at 4243(3)/1031.
Entry name "INIT_LINE" not found while attempting to resolve
dynamic link from procedure "TRY_ASYNC" .
ER!
```

Here, INIT_LINE is the target subroutine that is not known to PRIMOS, 4242/1031 is the address of the instruction that referenced a faulted IP for INIT_LINE, and TRY_ASYNC is the name of the procedure making the reference.

Note

PRIMOS is not always able to determine the name of the procedure making the reference that produces the linkage fault error. For example, procedures compiled in FTN do not identify themselves to PRIMOS; therefore, PRIMOS produces a shorter message. For example:

```
Error: condition "LINKAGE_FAULT$" raised at 4347(3)/10246.
Entry name "GETLIN" not found.
ER!
```

HOW DOES DYNAMIC LINKING RELATE TO COMMON BLOCKS?

PRIMOS does not support dynamic linking to common blocks; dynamic linking to subroutines only is supported. The BIND subcommand ENTRYNAME -ALL applies only to subroutines; ENTRYNAME -ALL does not declare common areas in a library EPF as externally available entrypoints.

> **Caution**
>
> Do not explicitly name a common area in a DYNT or ENTRYNAME subcommand in BIND, or you may encounter unexpected results.

3
The EPF Mechanism

This chapter describes the EPF mechanism itself. You should read this chapter if you want to more thoroughly understand how EPFs are handled by PRIMOS.

There are three general areas with which the EPF mechanism is concerned:

- Memory allocation
- Subroutine linkage
- Data initialization

In terms of these three general areas, this section describes:

- The organization of an EPF
- The organization ot subroutines on Prime systems
- The life of an EPF
- How multiple invocations of an EPF are handled
- How simultaneous use of an EPF by two or more users is handled
- How to debug an EPF using DBG
- How invocation of a remote EPF is handled

EPF ORGANIZATION

A running EPF is organized into five basic sections:

- Procedure text
- Linkage text
- Stack space
- Dynamically allocated memory (optional)
- Debugger text (optional)

Not all of the information in these five sections exists before the EPF is invoked. The file containing an EPF, named program.RUN or program.RPn, contains only the following information:

- Procedure code
- Linkage text initialization information
- Stack allocation information (included in linkage text)
- Debugger text

Procedure Code

The procedure code for an EPF is the most stable aspect of an EPF. Once a procedure is compiled or assembled, the contents of the procedure code are set. Once an EPF is linked by BIND, the pure procedure segments of that EPF are set and are not changed. An EPF may also have impure procedure segments; the contents of an impure procedure segment may be changed by PRIMOS or by the program itself as it runs. Typically, a procedure segment is impure because it contains linkage data (such as ECBs or IPs), although it may be impure because the code actually modifies itself as it executes.

Linkage Text

The linkage text for an EPF is the most complex aspect of an EPF. Linkage text includes:

- Program data
- ECBs for subroutines
- Links between subroutines
- Links to common data areas
- Common data areas

The final content for this information is determined at different points in time during the life of an EPF, depending upon the nature of the information. Because all of this information resides in the impure part of an EPF, and most of it resides in the Linkage text of an EPF, various portions of the linkage text of an EPF are finalized at different points in time during the life of that EPF. Therefore, the file containing an EPF rarely contains the final content of the linkage text; instead, it contains a combination of final information and information on how other portions of the linkage text are to be initialized.

Although common data areas are, themselves, not part of the linkage for any particular procedure, BIND places them in the linkage portion of an EPF along with the linkage text.

Stack Space

As noted, the stack allocation information is stored in the linkage text. Stack is allocated automatically during the invocation of a subroutine via the PCL (Procedure Call) instruction; the ECB for a subroutine includes the amount of stack space to be allocated each time the subroutine is called. Therefore, stack allocation is performed each time a subroutine is called.

Dynamically Allocated Memory

Dynamically allocated memory is acquired during program execution as a result of explicit requests by the procedure code of the program.

Because both stack storage and dynamically allocated memory are acquired during program execution, the file containing an EPF does not specify initial values for data within those areas. In fact, data in stack storage or in dynamic storage must be initialized during program execution.

Debugger Information

Information provided via the -DEBUG option of most Prime-supplied compilers is also kept in an EPF. DBG alone uses this information during the debugging of an EPF. You do not need to be concerned with the nature of this information, as long as you understand that it is passed from the compilers to BIND via object (.BIN) files, and from BIND to DBG via the EPF.

SUBROUTINE ORGANIZATION

The Prime 50 Series architecture defines a subroutine, or procedure, as consisting of:

- Procedure code
- Linkage text
- Stack space
- An Entry Control Block (ECB) that contains information on the above three elements

A particular procedure is said to consist of a procedure frame, a link frame, and a stack frame, which correspond to the terms listed above.

Management of the elements of a procedure is handled at runtime by the PCL and PRTN instructions. To call a procedure, a PCL instruction that addresses, as its target, the ECB of the desired procedure, is executed. The PCL instruction analyzes the ECB for the procedure being called, allocates a stack frame, and handles the transfer of arguments between the procedures. Before the PCL instruction completes, it sets up three base registers (PB for Procedure Base, LB for Linkage Base, and SB for Stack Base) pointing to the three frames listed above. These base registers are used by the procedure being called during its execution to execute instructions and access data in its linkage and stack areas.

A procedure therefore consists of only one copy of its procedure code and link frame, whereas it has any number of stack frames, depending upon the number of active invocations of that procedure. The EPF mechanism, however, provides a method for maintaining more than one copy of the link frames for an entire EPF when appropriate. Each separate invocation of a program EPF is given its own copy of its link frames by the EPF mechanism; similarly, each use of a program-class library EPF by a separate program invocation is given its own copy of its link frames. (See Chapter 6 for information on library EPFs.)

At the beginning of each stack frame is a stack header that contains the information needed to identify the owning procedure and also the information needed to return to the calling procedure. The stack header may also contain information on conditions and on-units to be invoked; this information is defined and set up by software on the Prime 50 Series systems.

When the called procedure is finished, it executes the PRTN instruction. The PRTN instruction deallocates the stack frame used by the procedure, resets the three base registers to their original values (before execution of the matching PCL instruction), and returns control to the calling procedure at the instruction following the PCL instruction (and its argument list, if any).

Typically, the linkage information for a procedure also contains the ECB for that procedure. It may seem strange that the ECB for a procedure not only identifies the address of the link frame of that procedure but also resides in the same link frame. It is the calling procedure that must address the ECB of the target procedure; the calling procedure does this by specifying an indirect pointer, or IP, that resides in its own link frame. This IP is set by either BIND or PRIMOS to point to the target ECB. Because an IP is a full address, rather than an address relative to the PB, LB, or SB base registers, the ECB may reside in the link frame of the target procedure.

General information on the procedure call mechanism is found in the Prime 50 Series Technical Summary. Details of the procedure call mechanism, including formats of the ECB and stack frame, are found in the System Architecture Reference Guide.

THE LIFE OF AN EPF

In the following discussion, the life of an EPF is presented as a series of phases through which an EPF passes from generation by the programmer, through invocation by a user or program, through termination, to removal from memory. Each of the above three areas is touched upon during the description of the phases.

In very general terms, the life of an EPF can be seen as progressing through five stages:

1. Generation by the programmer
2. Invocation (by a user or by a program)
3. Preparation by PRIMOS
4. Execution
5. Removal by PRIMOS

Each of these stages consists of one or more phases that, when put together, constitute a complex series of steps in the life of an EPF. Some of these phases differ in meaning between types of EPFs. For example, a program EPF is invoked directly by a user or program, whereas a library EPF is invoked as a result of a call to one of its entrypoints by another running program.

Prime-supplied text editors and compilers, the Prime Macro Assembler (PMA), and the BIND linker perform the activities that constitute Stage 1 at the direction of the programmer. After Stage 1, the EPF is in the hands of its users and is operated on by PRIMOS.

As described in Volume III of this series, there are many PRIMOS-supplied subroutines that operate on an EPF. One subroutine, EPF$RUN, performs all of the tasks needed to invoke a program EPF.

Invocation of an EPF, whether by EPF$RUN (for a program EPF) or by calling one of its entrypoints (for a library EPF), is Stage 2 in the above list.

EPF$RUN performs its tasks by calling other EPF$ subroutines to prepare the EPF for execution (Stage 3), to execute the EPF (Stage 4), and to remove the EPF from memory (Stage 5).

All five stages apply equally for both types of EPFs. Library EPFs, however, are not executed in the same way as program EPFs; instead, the subroutines they contain are executed as the calling program invokes them.

The other EPF$ subroutines called by EPF$RUN are called for every EPF run by PRIMOS. The exception is the EPF$INVK subroutine, which invokes a program EPF.

To describe the EPF mechanism faithfully, only those phases that are generally common to both program EPFs and library EPFs are presented here. Where appropriate, each step is correlated with the appropriate EPF$ subroutine. Chapter 6 describes aspects of the mechanism specific to library EPFs; Volume III of this series describes the EPF$ subroutines, including EPF$RUN, in detail.

There are ten phases of activity during the life of an EPF:

1. The procedures that are to constitute the EPF are compiled or assembled, generating object files (.BIN files).

2. These object files are linked using BIND, generating an EPF.

3. The EPF is invoked by either a user or a running program (EPF$RUN or the dynamic linking mechanism).

4. The procedure (pure) portion of the EPF is mapped to memory (EPF$MAP).

5. The linkage (impure) portion of the EPF is allocated (EPF$ALLC).

6. The linkage (impure) portion of the EPF is initialized (EPF$INIT).

7. The EPF entrypoint is invoked (EPF$INVK or the dynamic linking mechanism).

8. Dynamic links encountered within the EPF are snapped.

9. The EPF terminates, returning to its caller.

10. The EPF is removed from memory (EPF$DEL).

The remainder of this chapter describes these phases in detail.

Phase 1 - Compiling or Assembling

The source code of the program or library specifies the exact contents of the procedure text and the desired contents of the linkage text. The purpose of this phase is to generate an object file containing this information in a form that is independent of the language used in the source code.

Procedure text includes instructions and often includes constants used during execution of the procedure. Linkage text includes:

- Static data, which can be initialized when the program is invoked (via FORTRAN DATA statement or PL1/G STATIC INITIAL attributes)
- The ECB for the procedure and an ECB for each alternate entrypoint and for each internal procedure
- External linkage information, such as pointers to external procedures or to common areas

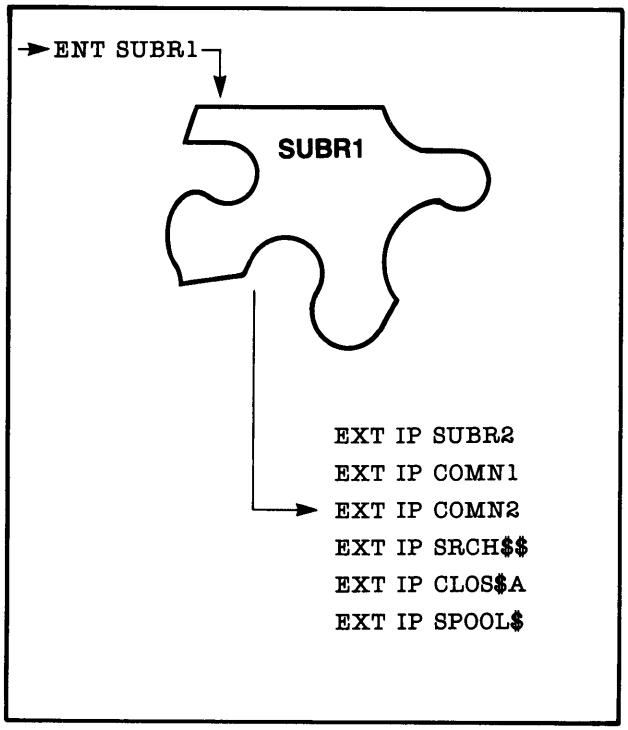
Static data values are known during this phase. Most of the data in an ECB are known during this phase except for the actual location of the procedure frame and the linkage frame for the procedure. The data for external linkage information are not known during this phase.

Data that are not known during this phase are described using alternate methods. For example, the statement CALL SUBR1 might reference an external procedure named SUBR1. The external linkage information indicates a requirement for an IP (Indirect Pointer) to SUBR1 at a certain location in the link frame. Either BIND or PRIMOS sets the data value of the IP. The procedure code, which is known during this phase, includes a PCL instruction referencing the location of the IP in the link frame as an indirect reference. Figure 3-1 illustrates linkage information as it exists in a single object (.BIN) file.

In Figure 3-1, the procedure SUBR1 references six external symbols: SUBR2, COMN1, COMN2, SRCH$$, CLOS$A, and SPOOL$. Figure 3-2 illustrates linkage information for the three object files, including SUBR1.BIN, that will be used to build a program EPF.

Phase 2 - Linking

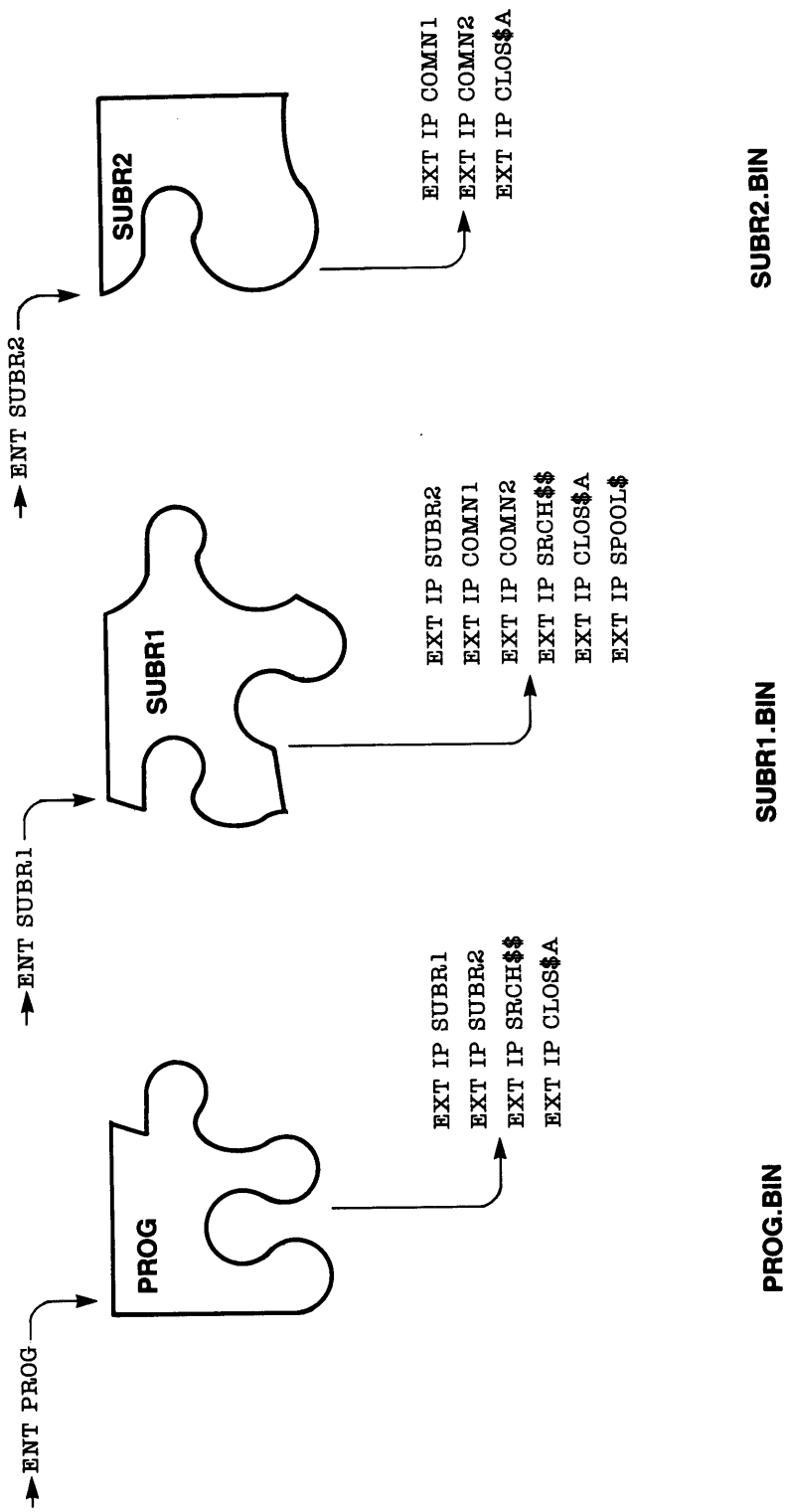
Object files are linked together using BIND during this phase. BIND maintains a list of procedure (PROC, or pure) segments and linkage (DATA, or impure) segments needed to represent the resulting program. The purpose of this phase is to produce an EPF file that contains the PROC, IMPURE, and DATA segments needed to run the program or library. Also, debugger information is written to the EPF file when this information is provided in the object files.



SUBR1.BIN

ENT is an externally available name declaration
EXT IP is an external (unresolved) Indirect Pointer

Linkage Information in a Single Object File
Figure 3-1



Linkage Information in Three Object Files
Figure 3-2

The procedure code from the linked object files is collected into one or more PROC and IMPURE segments. Link frames from these object files are collected into one or more DATA segments. Common areas referenced (and optionally initialized) by these files are also placed in DATA segments.

While collecting link frames, BIND maintains a list of external symbols and their placement in the program so that it can resolve some of the references in the link frames.

For example, if the CALL SUBR1 statement exists in a program named PROG, and the BIND session to create PROG.RUN links SUBR1 in with it, BIND resolves the IP to SUBR1 in the link frame of PROG to point to the SUBR1 ECB in the linkage text of SUBR1.

However, in most cases, BIND cannot resolve IPs to the actual memory addresses. Each time an EPF is invoked, PRIMOS dynamically allocates memory for the linkage text of the program. Therefore, BIND does not know the actual memory address of the linkage text of SUBR1, and therefore cannot compute the actual memory address of the SUBR1 ECB.

Instead, BIND leaves the task of determining actual memory addresses to PRIMOS. BIND identifies all IPs and ECBs that must be adjusted at program runtime by PRIMOS. In the meantime, BIND uses imaginary memory addresses. An imaginary address identifies a location within a particular EPF. When an EPF is invoked, PRIMOS maps the EPF into virtual memory and maintains a table indicating the mapping between imaginary and actual addresses.

In our example, therefore, BIND might put the linkage text for SUBR1 at imaginary address -0002/60. (Due to the architecture of Prime 50 Series computers, linkage text starts at octal 400 halfwords beyond the address in the base register that identifies linkage text for a procedure. Therefore, if linkage text is at -0002/60, the address used by BIND must be -0002/177460, because 60 minus 400, in octal and using 16-bit unsigned halfword arithmetic, is 177460.)

If the ECB for SUBR1 begins at octal 120 halfwords beyond the start of the linkage text, then the imaginary address for the ECB is -0002/200. BIND places -0002/200 in the linkage text location for the IP needed by PROG when it calls SUBR1 so that it uses the SUBR1 ECB as its target. BIND also places -0002/177460, the offset start of the SUBR1 linkage text, into the appropriate portion of the SUBR1 ECB. In both cases, BIND identifies that it has used imaginary addresses in the file containing the EPF, so that when the EPF is invoked, PRIMOS knows to modify these addresses.

BIND uses segment numbers +0, +2, +4, +6, +10, +12, and so on to indicate imaginary addresses identifying pure procedure (PROC) segments. BIND uses segment numbers -2, -4, -6, -10, -12, -14, and so on to indicate imaginary addresses identifying linkage (DATA) segments or impure procedure (IMPURE) segments. For more information on imaginary addresses vs. actual addresses, see Chapter 9.

Figure 3-3 illustrates the resolution of some of the external symbol references made by the three object files after BIND is invoked and the three files are linked via the LOAD subcommand.

As illustrated in Figure 3-3, external references to SUBR1 and SUBR2 have been resolved to the extent that their placement relative to the start of the EPF itself is known. There is only one way to produce actual memory addresses in BIND, and that is by using the SYMBOL subcommand.

When you use the SYMBOL subcommand to tell BIND exactly where an external symbol is located in memory, BIND does not use an imaginary address. Instead, it replaces all of the program's unresolved IPs that reference the external symbol with the actual address you specified using the SYMBOL command. Because use of the SYMBOL command requires you to manage actual memory, its use is generally confined to specifying shared common areas, as described in Chapter 8.

The SYMBOL subcommand may also be used to specify a common area in static per-user memory. Figure 3-4 illustrates this use of the SYMBOL subcommand.

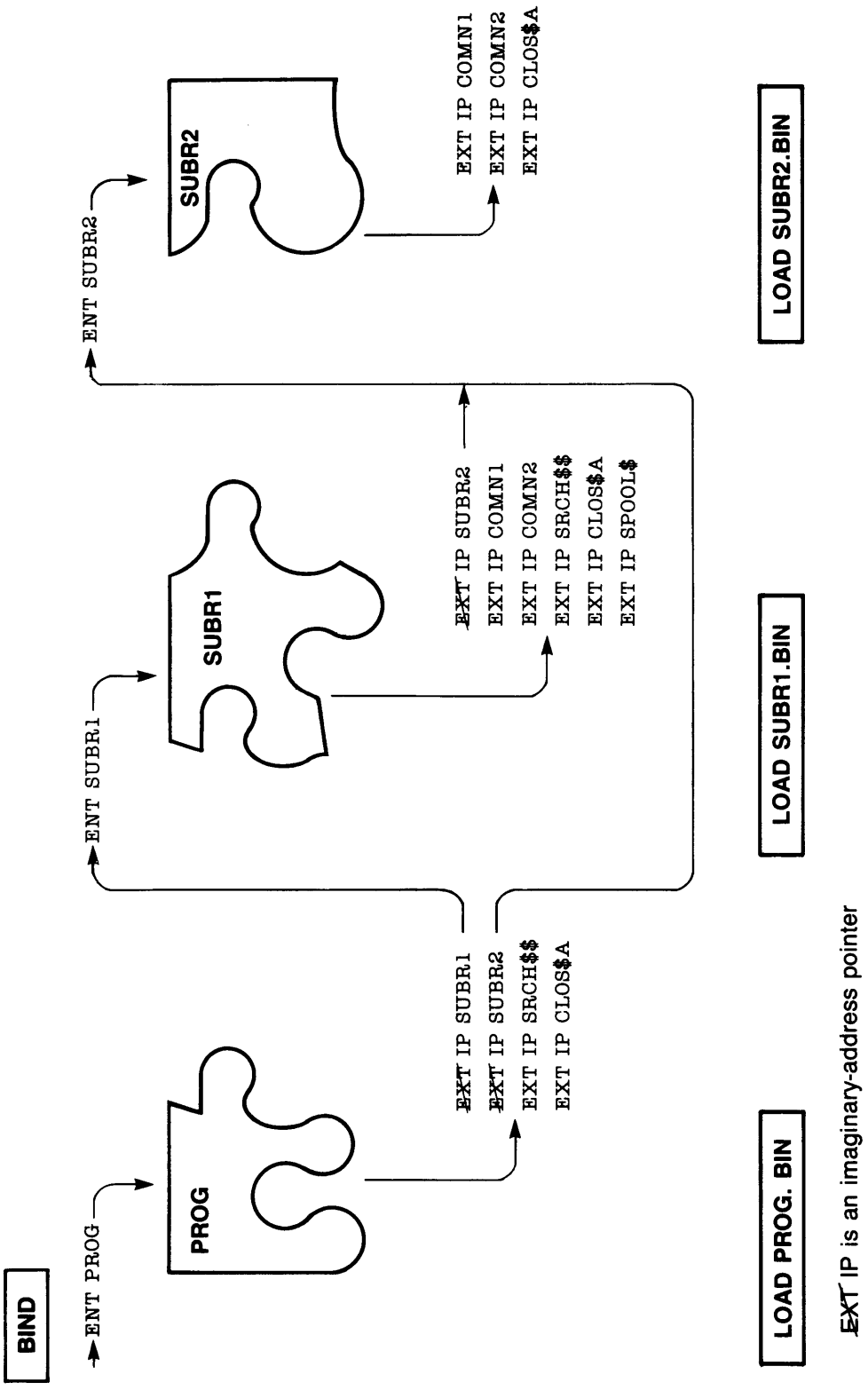
In Figure 3-4, the SYMBOL subcommand is used to specify that the common area named COMN2 is placed at address 4001/0, which is in static per-user memory. BIND therefore fully resolves all external references to COMN2 to actual memory addresses.

Note

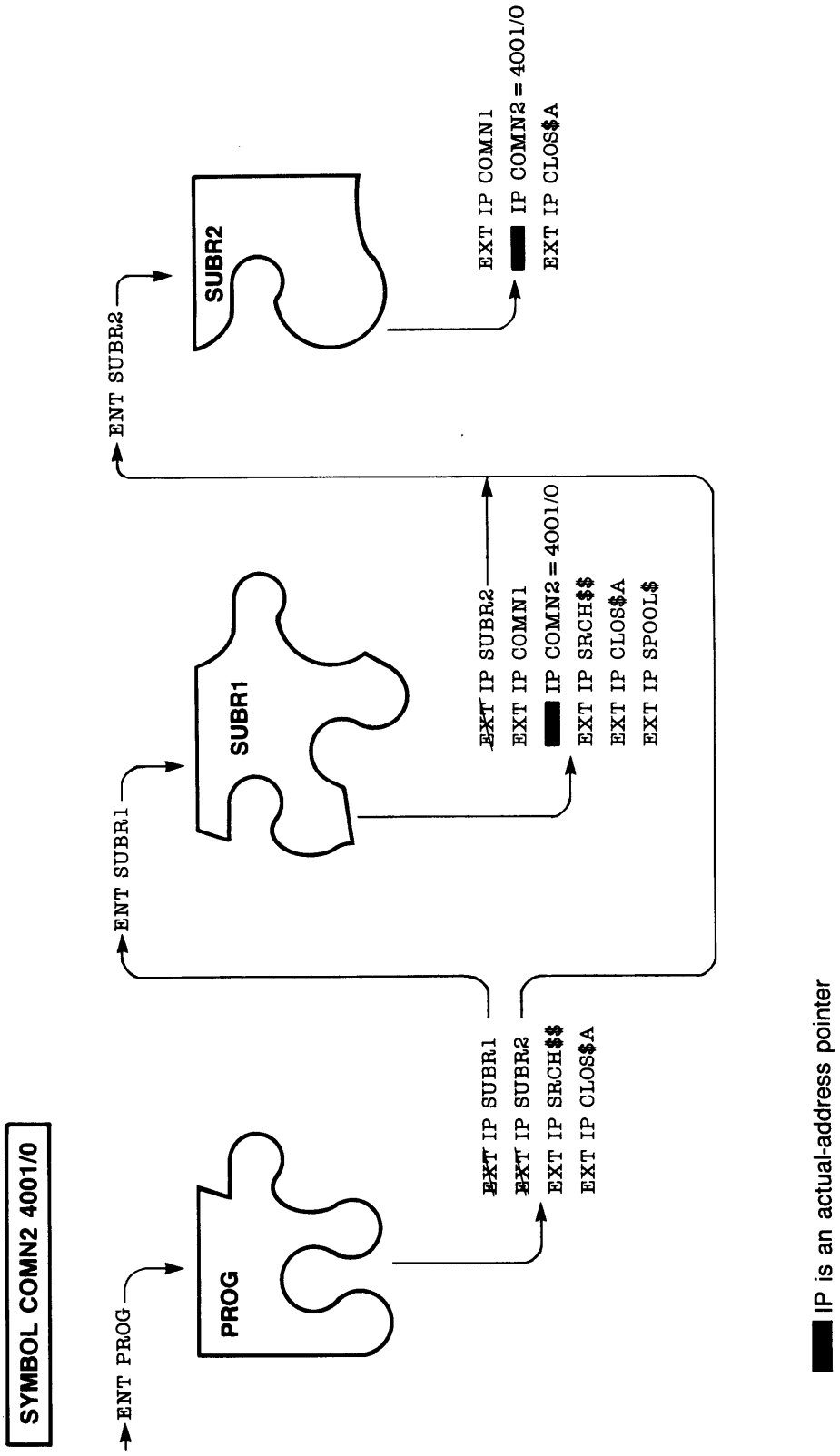
> Common areas that are placed by the SYMBOL subcommand of BIND are not initialized by BIND during program linking or by PRIMOS when the program is invoked. Either the program must initialize the common area at runtime, or it must be preinitialized by another program (as is often the case when shared memory is involved).

External references still unresolved include references to SRCH$$, CLOS$A, and SPOOL$. These are Prime-supplied subroutines that reside in libraries. These references are resolved when the libraries are linked via the LIBRARY subcommand of BIND. However, they are resolved only to the extent that they are identified as dynamic entrypoints, also known as dynamic links. IPs to these subroutines are converted to faulted IPs by BIND, as illustrated in Figure 3-5.

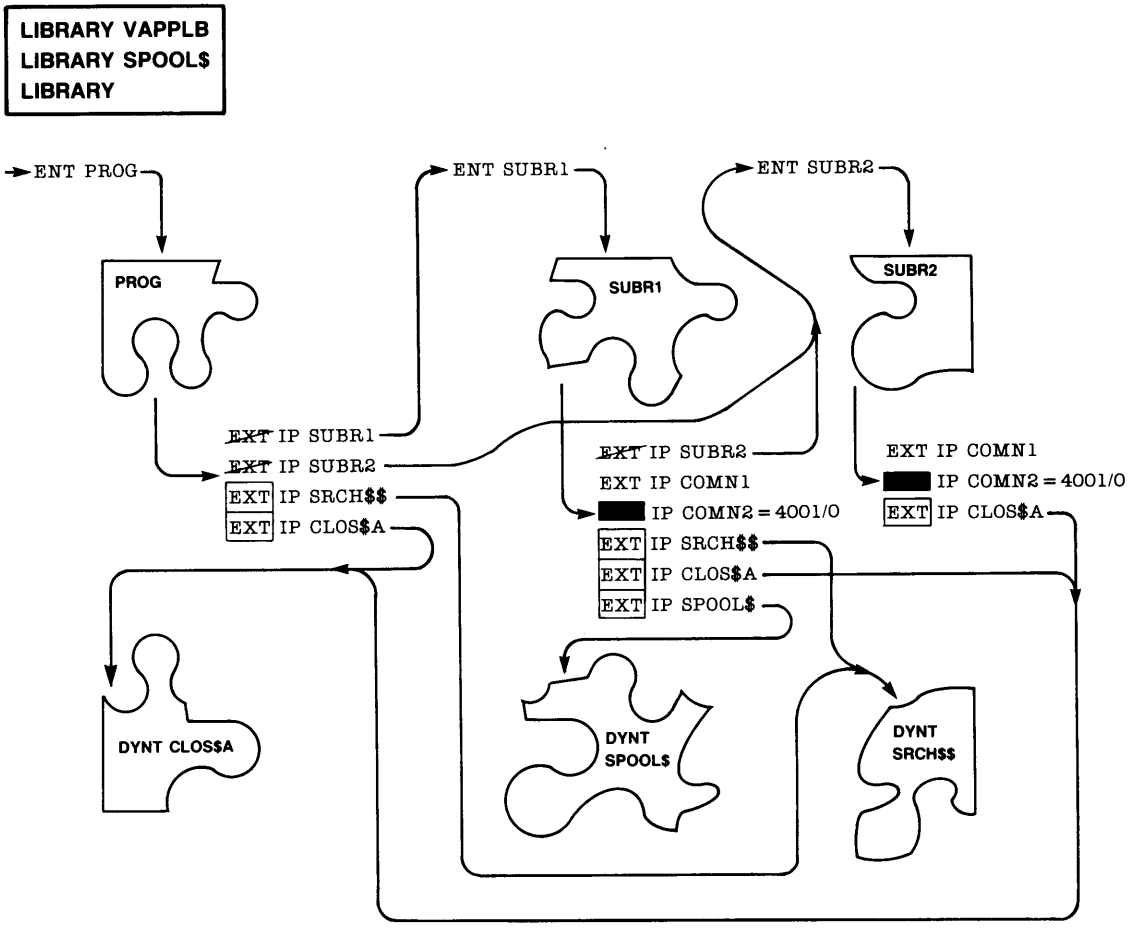
A faulted IP has a bit, called the fault bit, set to 1. This causes a hardware fault to take place whenever the IP is referenced as an indirect pointer, as described in Chapter 2. With the fault bit in the IP reset to 0, the IP points to a DYNT that names the desired subroutine. Because this name is in the PROC code of an EPF, BIND must use an imaginary address in the faulted IP as it does for other addresses not set using the SYMBOL subcommand.



After Linking (Loading) Three Object Files in BIND
Figure 3-3



After Using the SYMBOL Subcommand in BIND
Figure 3-4



EXT IP is a faulted pointer.
DYNT is a dynamic-link declaration.

After Using LIBRARY Subcommands in BIND
Figure 3-5

There still remain two unresolved references to the COMN1 common area. BIND knows the size of the common area, because this information was provided in the object files that referenced the common area. However, BIND has not yet resolved the two external IPs to COMN1, because it does not yet know whether COMN1 is part of the EPF, or external to the EPF (as defined by the SYMBOL subcommand) as is COMN2. Once the FILE subcommand is given, BIND resolves the dilemma by placing any unresolved common areas within the LINK portion of the EPF and resolving IPs to the common areas to imaginary addresses.

However, if you wish to produce a MAP of an EPF before issuing the FILE subcommand, you may want the imaginary addresses produced for such common areas to be present in the map. To effect this, issue the RESOLVE_DEFERRED_COMMON subcommand of BIND before issuing the MAP subcommand of BIND. Figure 3-6 illustrates the effect of the RESOLVE_DEFERRED_COMMON subcommand on IPs to COMN1 and on the EPF itself.

As shown in Figure 3-6, BIND has allocated storage in the EPF for the COMN1 common area, and resolved IPs to COMN1 so that they point to this area. Although not shown, COMN1 is located in the LINK portion of the EPF. The IPs to COMN1 are imaginary addresses, because the final address for COMN1 is not known until after the EPF is invoked.

Although all external IPs have been resolved in some fashion, there remains the task of defining the entrypoint or entrypoints to the EPF. A program EPF has only one entrypoint, defined by the MAIN subcommand of BIND, as illustrated in Figure 3-7. A library EPF has one or more entrypoints, defined by the ENTRYNAME subcommand of BIND, as described in Chapter 6.

Figure 3-7 illustrates the final EPF file produced by a FILE subcommand after the entrypoint(s) of the EPF have been identified. PRIMOS uses the MAIN or ENTRYNAME entrypoints when invoking the EPF. If no MAIN subcommand is given while linking a program EPF, BIND uses the first module linked via the LOAD subcommand as a default. If no ENTRYNAME subcommand is given while linking a library EPF, the library EPF is effectively useless because the dynamic linking mechanism will not link to any subroutines contained in it.

Figure 3-7 also illustrates that, once the EPF is written to a file, the boundaries between procedures are less distinct; BIND assembles the procedure frames and link frames for the procedures into single procedure segments and link segments for use by PRIMOS. Generally, PRIMOS is not aware of the boundaries between procedures within an EPF; PRIMOS is more concerned with the boundaries between EPFs in memory.

Phase 3 - EPF Invocation

When an EPF is invoked, PRIMOS begins the process of preparing the EPF so that it can be executed. Portions or all of this process may be avoided, however, if they have been performed previously.

The form of invocation depends upon the type of the EPF. A program EPF is invoked by calling the EPF$RUN subroutine or by calling the CP$ subroutine to execute an EPF. (CP$ is called to process a user command such as RESUME; it calls the same EPF$ subroutines that EPF$RUN calls.) A library EPF is invoked by a program that references one of its entrypoints by name.

Once PRIMOS has determined the name of the EPF to be invoked, it checks to see if it has already performed some or all of the preparatory steps needed to execute the EPF. For a program EPF, PRIMOS considers whether it must map the EPF to memory; if it has already mapped the EPF, it skips Phase 4. A program EPF is already mapped to memory if it is either on the EPF cache or if another invocation of the same EPF is still active on the command stack, such as when it is suspended by the user.

For a library EPF, PRIMOS has already mapped in the EPF at least to check its list of entrypoints to see if the desired subroutine is an entrypoint in the EPF. A library EPF is already mapped to memory, therefore, by the time PRIMOS determines that the EPF is the target of a dynamic link. PRIMOS then determines whether Phases 5 and 6 may be skipped. See Chapter 6 for a complete description of how PRIMOS decides whether to skip phases 5 and 6 for a library EPF.

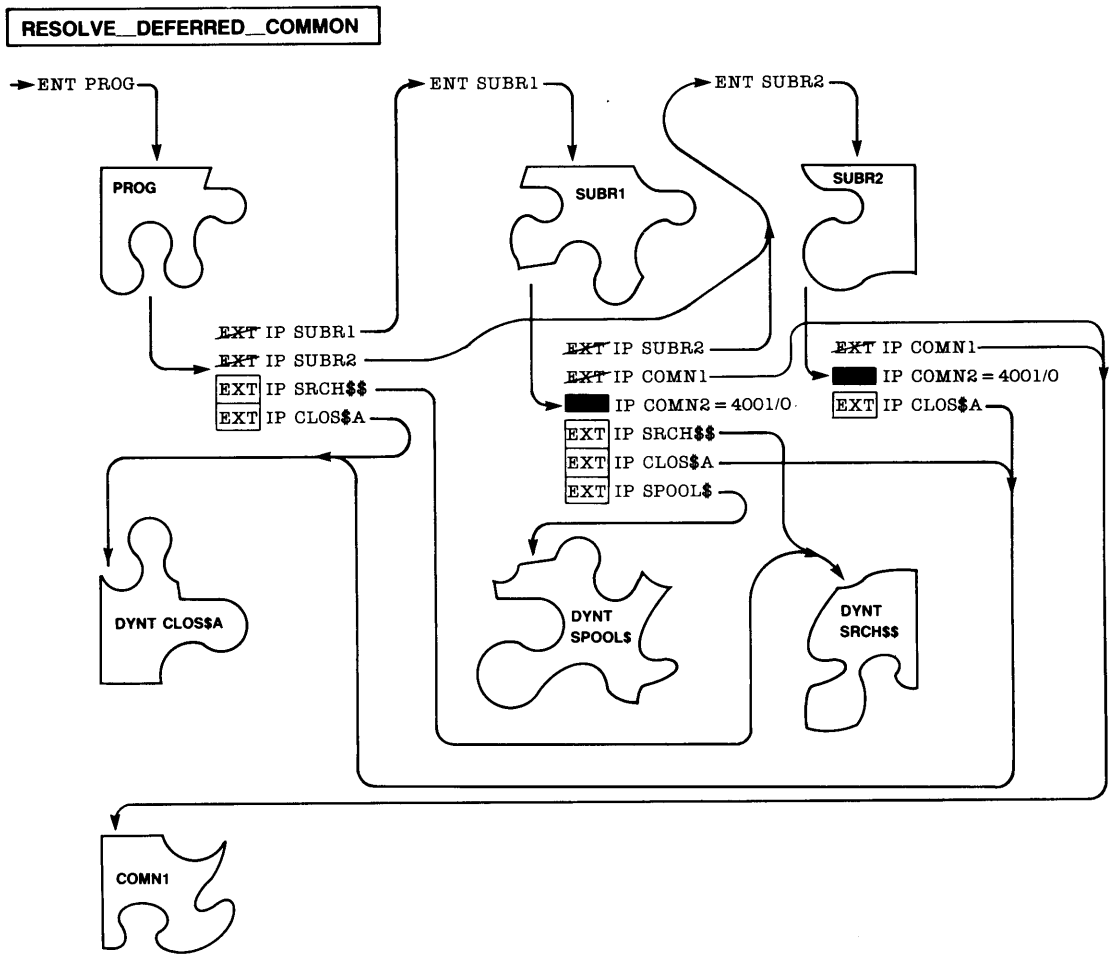
Phase 4 - Mapping

During Phase 4, PRIMOS allocates sufficient dynamic segments to hold all of the pure procedure (PROC) segments required by the EPF. The EPF$MAP subroutine performs the tasks associated with this phase. Information on space requirements for procedure and linkage segments is found by PRIMOS in the file containing the EPF.

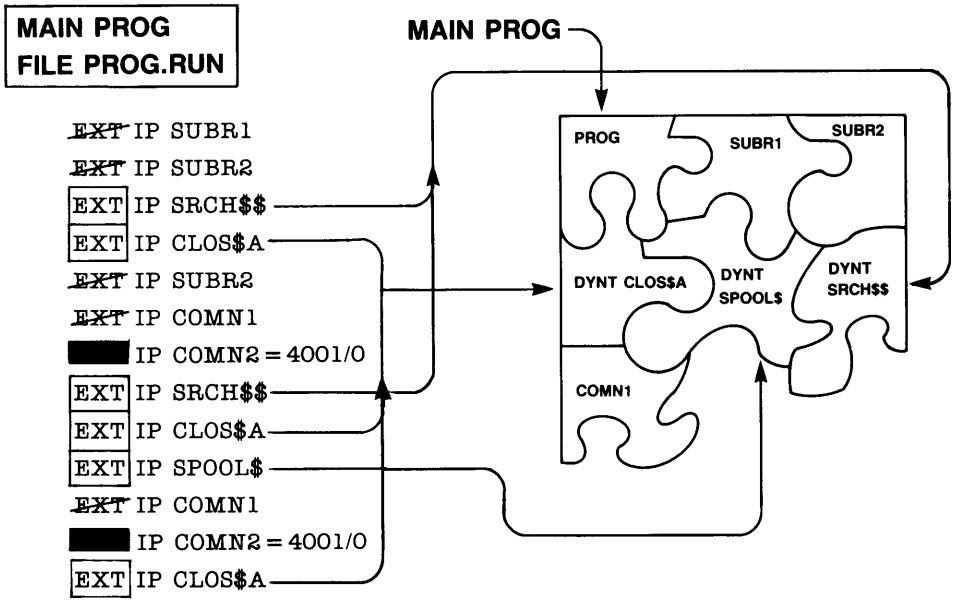
PRIMOS maps the PROC segments that have been allocated to the imaginary PROC segment numbers (+0, +2, +4, and so on) used in the file containing the EPF. In fact, PRIMOS does not read the procedure text in from the file at this point — instead, PRIMOS uses the virtual memory mechanism of PRIMOS to page data for these segments directly from the file containing the EPF. Because the virtual memory mechanism does not allow for segments mapped in this way to be modified, PRIMOS sets the access on these segments so that they cannot be written by the user or the program itself.

Phase 5 - Linkage Allocation

During Phase 5, PRIMOS allocates sufficient dynamic space to hold all of the linkage (DATA) segments and impure procedure (IMPURE) segments required by the EPF. PRIMOS sets the access on these segments so that they can be written by the user or by the program. The EPF$ALLC subroutine performs these tasks.



After Using the RESOLVE_DEFERRED_COMMON Subcommand in BIND
Figure 3-6



Program EPF PROG.RUN

After Using the FILE Subcommand in BIND
Figure 3-7

Phase 6 - Linkage Initialization

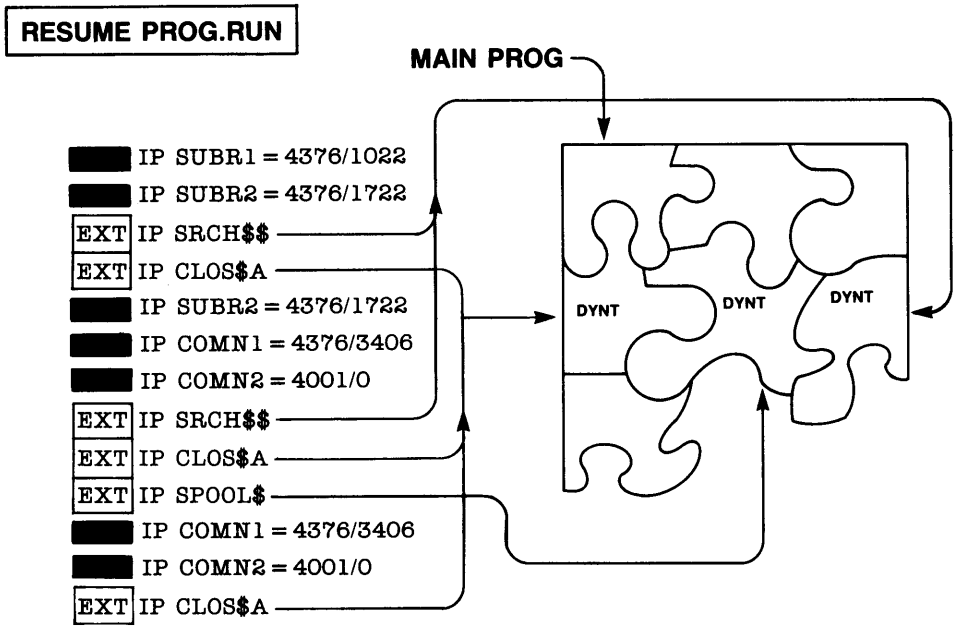
During Phase 6, PRIMOS reads the descriptor information for the linkage areas and impure areas of the EPF and sets the DATA and IMPURE segments that have been allocated to contain the indicated information. Data such as program constants are copied directly into the DATA segments. Linkage data consisting of imaginary addresses are first converted into the corresponding actual addresses for this program invocation, and are then copied into the DATA segments. This linkage data consists mostly of IPs and ECB linkage area pointers. Impure procedure code is copied directly into the IMPURE segments. These tasks are performed by the EPF$INIT subroutine.

Figure 3-8 illustrates a program EPF after its imaginary addresses have been converted to actual addresses.

Although Figure 3-8 illustrates the overall effect of a RESUME command on a program EPF just before its main entrypoint is actually invoked, the same actions are performed on a library EPF. All of the linkage information for both program EPFs and library EPFs is initialized, converting imaginary addresses to actual addresses for both IPs and ECBs. The only IPs that remain unresolved are faulted IPs. Faulted IPs are adjusted so that they contain the actual addresses of DYNTs, rather than the imaginary addresses produced by BIND.

Phase 7 - Entrypoint Invocation

At this point, PRIMOS either executes a PCL instruction to the ECB of the main entrypoint for a program EPF (performed by the EPF$INVK subroutine), or resets the fault condition and reexecutes the original PCL instruction that now references the ECB of the desired entrypoint for a library EPF (performed by the dynamic linking mechanism). The EPF is now running.



After EPF Is Mapped and Initialized
Figure 3-8

Phase 8 - Dynamic Links Snapped

During the execution of the EPF, faulted IPs are typically encountered. These represent dynamic links that must be snapped by PRIMOS before the instructions referencing the faulted IPs can be executed. Once snapped, a faulted IP becomes an actual memory address, and subsequent use of that particular IP produces no fault condition.

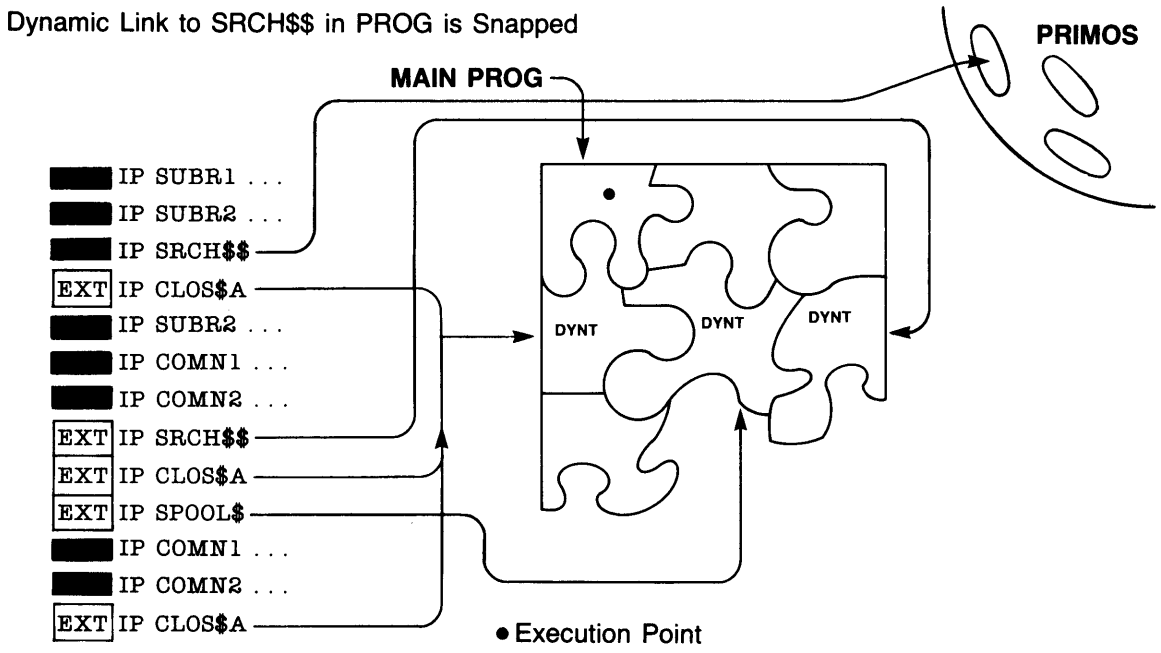
Because several different faulted IPs may point to the same DYNT within an EPF, several faults may resolve to the same subroutine within an EPF. In addition, while some entrypoints reside in PRIMOS, others reside in library EPFs or in static-mode libraries. To show the details of how dynamic links are snapped in an executing EPF, several illustrations are provided that follow the PROG program EPF through its execution until it has snapped all of its faulted IPs.

In practice, few programs ever snap all of their faulted IPs, because some faulted IPs point to error-handling subroutines (such as ERRPR$ and SIGNL$), while other faulted IPs point to subroutines used when no errors occur (such as TNOU, PRWF$$, and CLOS$A). Furthermore, a library EPF rarely snaps all of its links when it consists of more than one entrypoint, unless each entrypoint in that library EPF is invoked by the same calling program during its execution.

Figure 3-9 illustrates the PROG program EPF after the first dynamic link to SRCH$$ in the PROG procedure is snapped. Here, PRIMOS itself is illustrated as residing in the same memory space as the user program and containing many entrypoints, one of which is named SRCH$$.

In Figure 3-9, the first faulted IP in the EPF, which resides in the link frame for PROG, has been snapped. It has been replaced with the absolute address of the ECB of SRCH$$ in PRIMOS. Note that another faulted IP to SRCH$$, in the link frame for SUBR1, has not been resolved.

Dynamic links to entrypoints in PRIMOS itself are easy for PRIMOS to resolve. PRIMOS is always present in every user's memory address space. In addition, all PRIMOS entrypoints have their link frames initialized at system coldstart or during system build, so no initialization is needed.



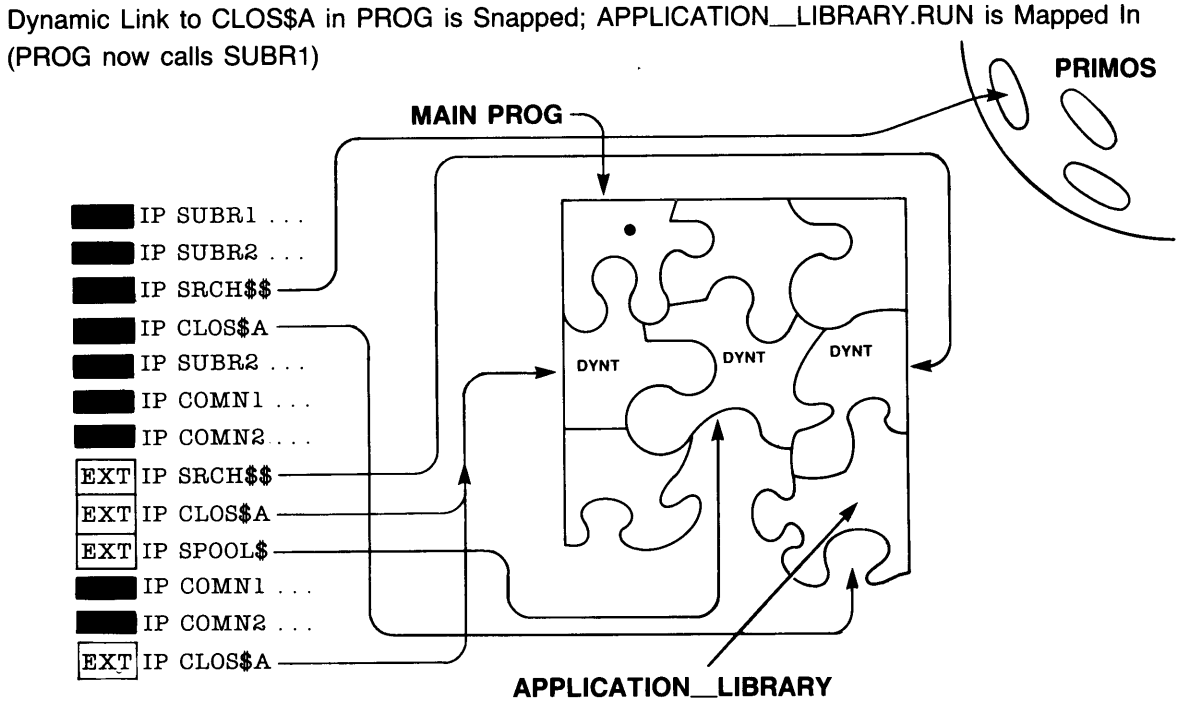
After First Dynamic Link Is Snapped
Figure 3-9

Next, a faulted IP to CLOS$A in the PROG procedure is snapped. CLOS$A is an entrypoint in the Application Library (described in the Subroutines Reference Guide), which is contained in the library EPF file LIBRARIES*>APPLICATION_LIBRARY.RUN. Figure 3-10 illustrates the appearance of the PROG program EPF after the dynamic link to CLOS$A is snapped.

As shown in Figure 3-10, the APPLICATION_LIBRARY library EPF has been loaded into memory and "connected" to the running program EPF PROG. In fact, by the time the first call to CLOS$A by the PROG procedure was completed, the library EPF named APPLICATION_LIBRARY went through Phases 3 through 9 as described here, when the CLOS$A entrypoint returned to its caller, the PROG procedure.

Again, notice that two other faulted IPs to CLOS$A, in SUBR1 and in SUBR2, have not yet been resolved.

Sometime after calling SRCH$$ and CLOS$A, the PROG procedure calls the SUBR1 procedure, which will encounter its own faulted IPs.



After Second Dynamic Link Is Snapped
Figure 3-10

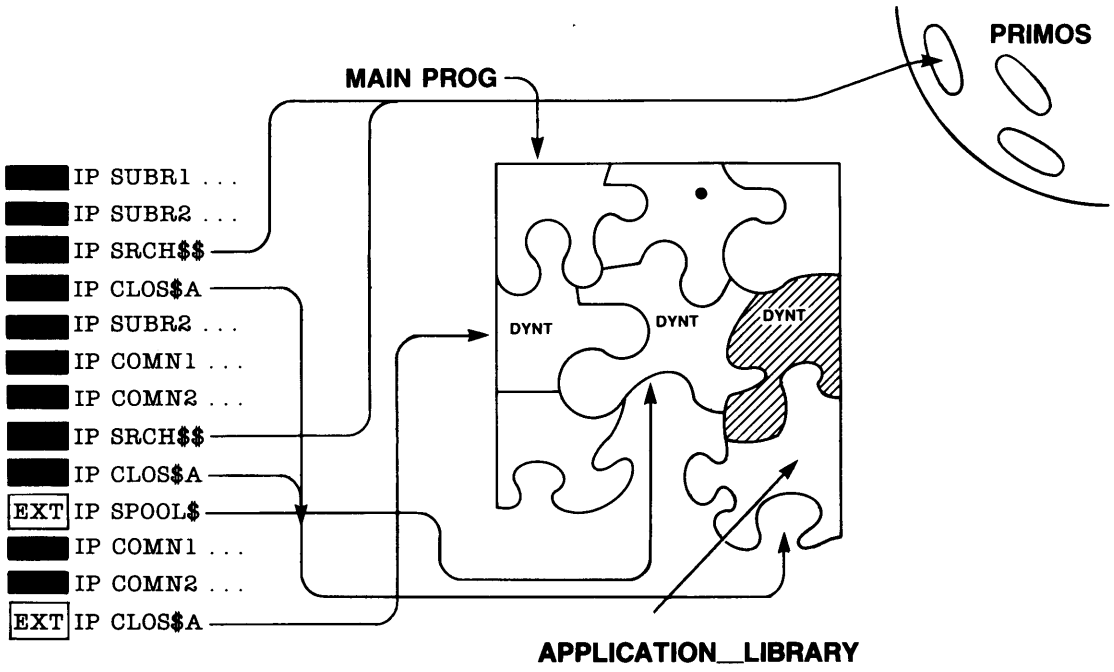
When dynamic links that identify entrypoints in libraries already initialized are encountered, PRIMOS needs to do much less work to make the connection. This is illustrated in Figure 3-11, in which the dynamic links to SRCH$$ and CLOS$A in SUBR1 are snapped.

Because SRCH$$ is an entrypoint in PRIMOS, the process of snapping the faulted IP to it is again straightforward. PRIMOS is already present in the user's memory area, as always, and no initialization of any PRIMOS-resident subroutines is ever needed at runtime.

Because CLOS$A is an entrypoint in a library EPF that has already been connected to the PROG program EPF, no initialization of the library EPF is needed. Even if a different entrypoint in the same library EPF, such as NLEN$A, had been called, no initialization would have been needed.

As there are no longer any faulted IPs pointing to the DYNT for SRCH$$, the DYNT in the illustration has been shaded in to indicate that it is no longer in use by that program invocation. This shading does not represent any action by PRIMOS; the DYNT still remains in the PROC segment as before. The shading is present only to indicate the absence of faulted IPs to the SRCH$$ DYNT.

Dynamic Links to SRCH$$ and CLOS$A in SUBR1 Are Snapped



After Third and Fourth Dynamic Links Are Snapped
Figure 3-11

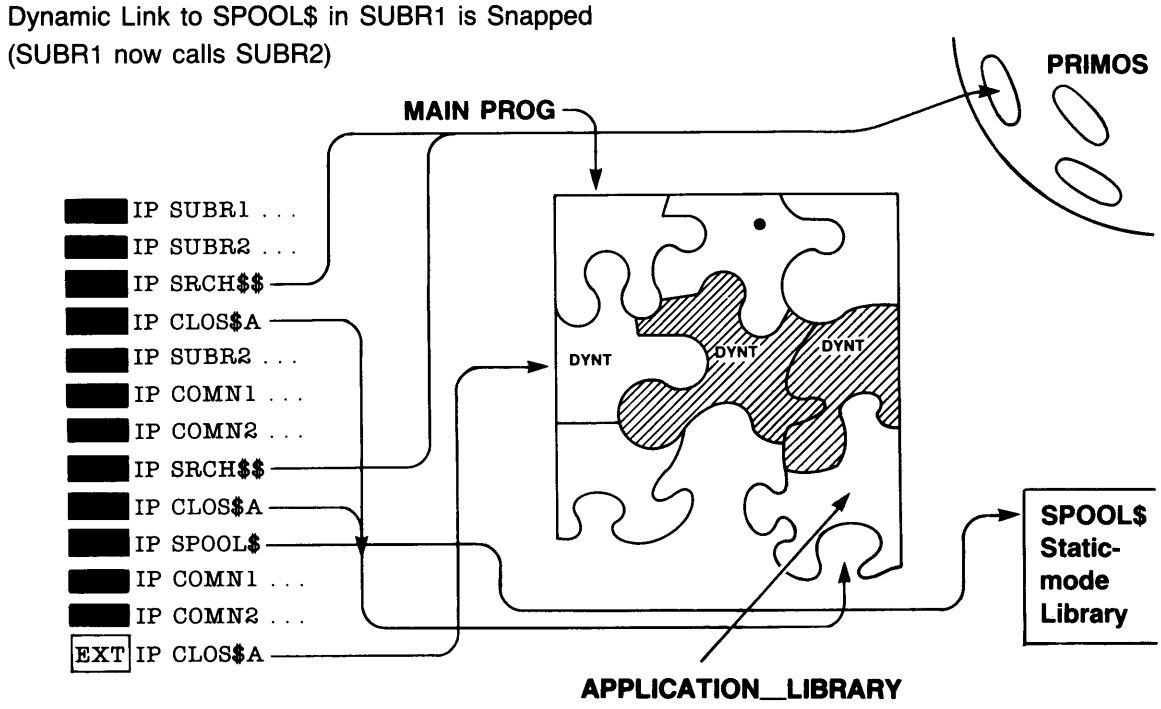
Next, the faulted IP to SPOOL$ in the SUBR1 procedure is encountered. SPOOL$ is a Prime-supplied entrypoint that comprises the program's interface to the Spooler subsystem. SPOOL$ is, at Rev. 19.4, supplied as a static-mode library. (Prime reserves the right to change SPOOL$ to a library EPF in the future.) Figure 3-12 illustrates the PROG program EPF after snapping the dynamic link to SPOOL$.

All static-mode libraries are placed in shared memory at system coldstart. In this manner, they have a similarity to PRIMOS entrypoints. Linkage information for a static-mode library is initialized when a program invokes that static-mode library for the first time. Here, a similarity to program-class library EPFs exists. Because linkage information for a static-mode library is statically located, only one active copy may exist for each user process. This indicates a similarity to process-class library EPFs.

In fact, once a program has invoked a static-mode library, any other suspended programs for the same user process that have used the same static-mode library are made unrestartable by PRIMOS. Any attempt to continue the execution of a suspended program that has already used a reinitialized static-mode library is thwarted by PRIMOS, which displays an error message. (See Chapter 4.)

As before, notice that the DYNT for SPOOL$ is now shaded, to indicate that the DYNT is no longer referenced by any faulted IPs in the PROG EPF.

The SUBR1 procedure now calls the SUBR2 procedure.



After Fifth Dynamic Link Is Snapped
Figure 3-12

Finally, the last faulted IP, to the CLOS$A subroutine, is encountered while executing procedure SUBR2. Figure 3-13 illustrates the PROG program EPF after snapping this final faulted IP.

As before, because the Application Library is already mapped into memory and initialized, only Phases 7 through 9 of the EPF process are involved in snapping this faulted IP. Notice that the last DYNT, to CLOS$A, has been shaded in to indicate that no more faulted IPs pointing to the CLOS$A DYNT exist.

The final picture of the PROG program EPF shows a program in which all DYNTs, which are placeholders for subroutines, have been replaced by the actual subroutines, whether they reside in PRIMOS, in library EPFs, or in static-mode libraries. From this point forward, such an EPF executes faster when it encounters snapped IPs, because the dynamic linking mechanism is no longer involved.

However, PRIMOS takes care to insure that these snapped IPs do not become a burden by preventing subsequent relocation of their targets. An in-use library EPF, such as APPLICATION_LIBRARY in the example, cannot be removed via the REMOVE_EPF command; if a new version is installed, the user running the PROG program EPF will not begin using the new version until after PROG has completed and the next program calls the Application Library. If another program invokes APPLICATION_LIBRARY while PROG is suspended, PRIMOS will allocate and initialize a separate linkage area for the library EPF so that a separate connection is made. This procedure preserves the integrity of the first connection made between PROG and APPLICATION_LIBRARY, because that connection will continue to use the originally allocated linkage area for APPLICATION_LIBRARY.

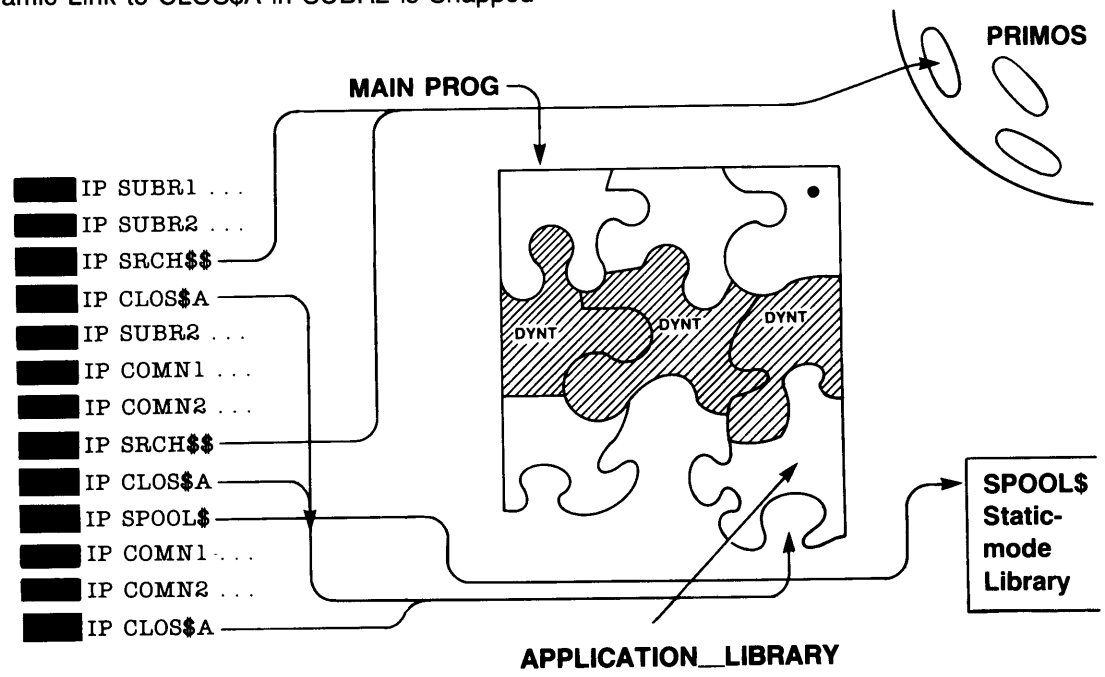
PRIMOS has less control over static-mode libraries because they are less flexible. No new version of a static-mode library may be installed while users are running programs that use it, or programs may stop functioning correctly. Instead, the installer should shut down and coldstart PRIMOS when installing a new version of a static-mode library to ensure that programs are not adversely affected.

If, while PROG is suspended, a static-mode library is reinitialized by another program's invocation of one of its entrypoints, PRIMOS detects this and flags PROG as being unrestartable. This prevents PROG from later calling the static-mode library and inadvertently using the linkage data in that library that was left over from the other program's use of that library.

Phase 9 - Termination

A program EPF terminates just once per invocation, either by returning from its main entrypoint (preferred) or by calling EXIT (an alternate way of terminating).

Dynamic Link to CLOS$A in SUBR2 is Snapped



After Final Dynamic Link Is Snapped
Figure 3-13

A library EPF terminates simply by returning from whatever entrypoint invoked it. A library EPF typically is invoked many times during the life of a program, and it accordingly terminates, or returns, many times. Unlike a program EPF, there is no point at which a library EPF really finishes, because it is actually a collection of subroutines that service a program.

Conceptually, a library EPF is part of the program that invokes it. When the program terminates, the library is also terminated by PRIMOS. However, PRIMOS distinguishes between two types of library EPF in this regard: program-class and process-class library EPFs. When a program that invoked a program-class library EPF terminates, PRIMOS terminates the library EPF by marking for reinitialization the copy of its linkage area it earlier allocated and initialized for use by the library EPF in service of that particular program.

Similarly, when a program EPF terminates, PRIMOS marks for reinitialization the linkage area used for that program invocation. When a linkage area is reinitialized, only program data and faulted IPs are actually reinitialized, saving startup time. (Linkage areas for program EPFs and program-class library EPFs are deallocated when the command level they were invoked from is released. In addition, linkage areas for program EPFs are deallocated when they are removed from the EPF cache.)

However, PRIMOS never reinitializes or deallocates the linkage area for a process-class library EPF when any program terminates, because that linkage area and the data it contains is usable for all programs invoked by that process. Only explicit removal (via REMOVE_EPF), logout (via LOGOUT), or command environment initialization (via INITIALIZE_COMMAND_ENVIRONMENT) causes PRIMOS to deallocate the linkage area associated with a process-class library EPF.

For program EPFs and program-class library EPFs, keep in mind that only one copy of the linkage area for a terminated EPF is deallocated. Other copies may exist if the same program EPF is suspended within the same process or if another suspended program is also using the same program-class library EPF. PRIMOS ensures that suspended program invocations are not affected by the termination of other programs.

PRIMOS also deallocates dynamically allocated memory (acquired via the ALLOCATE statement in PL1/G, for example) when a program EPF or program-class library EPF terminates. PRIMOS does not deallocate memory allocated by a process-class library EPF even when the REMOVE_EPF command is used to remove the EPF. Only the INITIALIZE_COMMAND_ENVIRONMENT and LOGOUT commands deallocate memory dynamically allocated by process-class library EPFs. See Chapter 6 for more information on storage allocation and library EPFs.

Phase 10 - Removal

Once an EPF has completely terminated — that is, once it is no longer in use by any suspended or executing program — PRIMOS may unmap the EPF from memory. The EPF$DEL subroutine performs this task. However, PRIMOS typically keeps an EPF mapped for future use, depending upon the type of EPF involved.

Removal of Program EPFs: PRIMOS usually places a terminated program EPF on the EPF cache rather than simply unmapping it. While on the EPF cache, the EPF is still mapped to memory. If the EPF is invoked again while on the EPF cache, the mapping step (Phase 4) is avoided. In addition, if the impure areas (such as linkage data) have not been deallocated, the allocation step (Phase 5) is avoided, and only a subset of the initialization step (Phase 6), reinitializing linkage information, is required.

If, on the other hand, the EPF cache becomes unwieldy, PRIMOS automatically unmaps the oldest EPF on the EPF cache, removing it from the EPF cache at the same time, and deallocating its impure areas.

When you change command levels, PRIMOS removes all program EPFs from the EPF cache, deallocating their impure areas and unmapping them from memory. Only suspended (in-use) program EPFs and all library EPFs remain mapped to memory. (This behavior may change at future Revisions of PRIMOS.)

Removal of Program-class Library EPFs: When all programs that are using a program-class library EPF have terminated, the library EPF is itself considered terminated. However, PRIMOS leaves the EPF mapped to memory and also leaves its impure areas (such as linkage data) allocated. This allows faster scanning of the entrypoint table for that library EPF in the future.

When you change command levels, PRIMOS deallocates all impure areas for program-class library EPFs that are not in use by a suspended or running program, but it leaves the library EPFs mapped to memory. (This behavior may change at future Revisions of PRIMOS.)

Removal of Process-class Library EPFs: When all programs that are using a process-class library EPF have terminated, the library EPF is itself considered terminated. However, PRIMOS leaves the EPF mapped to memory and also leaves its impure areas allocated because, once allocated and initialized, a process-class library EPF is considered initialized for the duration of that process (that is, until the user logs out or the command environment is reinitialized).

When not in use, you may remove a process-class library EPF with the REMOVE_EPF command. However, you can never remove an EPF that is in use (executing or suspended).

Normally, you do not need to be concerned with the EPF cache or the maintaining of library EPFs performed by PRIMOS. These features are present as an optimization feature of PRIMOS. They do provide the ability to display useful information on a recently invoked EPF via the LIST_EPF command without having to specify the -NOT_MAPPED option.

However, the maintaining of inactive EPFs in memory also may produce unexpected side effects. For example, while an EPF is mapped to memory for one or more users, installation of a new version of that EPF results in the generation of a .RPn file containing the old version, even though no user may be running the old version or may even have the old version suspended. The transparent nature of PRIMOS' creation of .RPn files during COPY operations should keep this from being a problem.

Note

An EPF cannot be deleted if another user either is running that EPF, has suspended that EPF, or has that EPF mapped to memory. On the other hand, if only one user is using an EPF, and the EPF is not running nor suspended (that is, it is a program EPF on the EPF cache or a library EPF still mapped to memory), then that user can delete the EPF. The DELETE command automatically removes an inactive EPF if it is found to be in use during the first deletion attempt; after removing the inactive EPF, it retries the deletion.

A program that maps an EPF by calling EPF$MAP may also explicitly call EPF$DEL to unmap that EPF. A program that calls EPF$RUN may make use of a special key that indicates that EPF$RUN should call EPF$DEL after invoking the program EPF to unmap it rather than place it on the EPF cache. See Volume III of this series for more information on EPFMAP, EPFDEL, and EPF$RUN.

HOW MULTIPLE INVOCATIONS OF AN EPF ARE HANDLED

Sometimes a user might invoke an EPF, either explicitly by running a program EPF or implicitly by running a program that calls a library EPF, then suspend the EPF via Control-P, and issue a command that causes the same EPF to be reinvoked. Here, PRIMOS must allow the second invocation of the EPF to execute without disturbing the state of the first EPF, so that the user has the option of later continuing the first invocation.

When PRIMOS detects that an EPF that is already mapped into memory is being invoked, PRIMOS checks the type of the EPF. If the EPF is a process-class library EPF, PRIMOS reuses the already-initialized linkage area for that EPF. For other types of EPFs, program EPFs and program-class library EPFs, PRIMOS allocates new memory for the linkage areas of the EPF, rather than reusing any existing linkage areas for

the EPF. In this case, PRIMOS must initialize this newly allocated linkage information. Because the stack is preserved when a program is suspended, and because PRIMOS does not destroy the linkage of the suspended EPF, reinvoking the same EPF does not necessarily overwrite the data used by the suspended invocation. (See the restriction on using static-mode libraries, in Chapter 4, for an exception to this rule.)

However, to save time and memory, PRIMOS does reuse the same pure procedure (PROC) segments used by the first invocation of the EPF. PRIMOS can do this because segments mapped in this manner cannot be modified, and because no data in PROC segments needs to be adjusted to reflect imaginary-to-actual memory addresses.

HOW SIMULTANEOUS USE OF AN EPF IS HANDLED

When two or more users are running the same EPF, PRIMOS detects this and shares pure procedure (PROC) segments between the users. It can do this even when one user's actual segment numbers for the EPF are different from another user's actual segment numbers, because no segment numbers are stored in PROC segments.

For example, if two users are running the same EPF at the same time, one user might have imaginary PROC segment number +0 mapped into actual segment number 4365, while the other user might have imaginary segment +0 of the same program EPF mapped into segment number 4372. Segment 4365 for the first user and segment 4372 for the second both map to the same imaginary PROC segment in the file containing the EPF, reducing actual memory usage and paging overhead.

To prevent one user from being able to adversely affect the smooth operation of another user's program, PRIMOS protects PROC segments so that they cannot be written into when they are shared in this fashion.

HOW DEBUGGING OF AN EPF IS HANDLED

You can use DBG on a program EPF by issuing the DBG program command rather than the RESUME program command. You cannot, however, use DBG to debug a library EPF directly. To debug a library EPF, link the object files that comprise it into a program EPF image with a main entrypoint consisting of a small module that invokes or otherwise declares the existence of all of the desired entypoints in the library EPF. Then, use the DBG program command to debug the resulting program EPF.

DBG allows you to set breakpoints in the program EPF you are debugging, so DBG must be able to modify the pure procedure (PROC) segments of the program. Yet PRIMOS normally maps the PROC segments to their imaginary counterparts in the file containing the program EPF, setting the access to the PROC segments so that they cannot be written into. How can DBG

set breakpoints in the code when it cannot write over any procedure instructions?

To handle this, DBG specifically requests PRIMOS to not map PROC segments into memory. Instead, PRIMOS copies data from the imaginary PROC segments in the program EPF file into the actual PROC segments in memory. PRIMOS sets the access on these copied PROC segments so that they can be written into. In addition, PRIMOS does not share these PROC segments with any other users.

Finally, if the user suspends DBG and invokes the program EPF he or she is debugging using the RESUME command, PRIMOS allocates new PROC segments for this second invocation of the program EPF, preventing breakpoints set in DBG from affecting the RESUME invocation of the program EPF. When the RESUME invocation has finished, the user may type START and continue debugging the DBG invocation of the program EPF.

One of the effects of using DBG, therefore, is that attempts by the program EPF being debugged to modify itself generally go undetected. Such attempts cause access violation errors only when the program EPF is invoked using the RESUME command.

HOW RUNNING A REMOTE EPF IS HANDLED

If a remote EPF is invoked, either by a user typing RESUME program where program.RUN resides on a remote system disk, or by an entrypoint search rule referencing a remote library EPF, PRIMOS does not map the PROC segments to the remote file. Instead, the PROC segments are copied from the file into memory before execution begins, in a fashion similar to the way IMPURE segments are handled.

This is done so that a running remote EPF will not be affected should the remote system or network be shut down. If the PROC segments were mapped, then the effect on the running program of the remote system becoming unavailable would be unpredictable. In any case, the condition could probably not be trapped by the running program, because the code to handle such a condition would probably not be in physical memory. (The chances of error recovery code being frequently referenced tend to be small). If the recovery code is not in physical memory, then it would have to be paged in from the remote system, with which contact had been lost.

Therefore, due to the amount of time required to copy a large remote program EPF into memory, it is recommended that you provide local copies of EPFs on all systems in your network. By doing this, you take advantage of the faster memory mapping mechanism that is used for local EPFs.

4

EPFs and Static-mode Applications

Because of the dynamic nature of EPFs, their interaction with existing static-mode applications is fraught with considerations and restrictions. In general, if you have a large body of software that is already built with SEG, it is best to designate one or two applications as pilot cases for conversion before committing to a wholesale conversion. Once you have succeeded at converting the chosen pilot applications, and have developed corresponding expertise on conversion difficulties peculiar to your installation, you should then perform a wholesale conversion rather than a piecemeal conversion, in order to avoid excessive concern over the considerations and restrictions described in this chapter.

However, piecemeal conversion is sometimes the only feasible approach, and, in such cases, you should thoroughly understand the information presented in this chapter before you begin, so that you know what pitfalls may await you.

In many cases, conversion involves simply relinking an application using BIND. However, certain uses of static system-defined data in static-mode programs may no longer function or may not be appropriate when these programs are converted to EPFs.

Even if you do not intend to convert any applications to EPFs in the near future, you may have an existing application that requires modification to run on a Rev. 19.4 (or beyond) system because of the existence of library EPFs that contain system library subroutines. Such an application is one that resides in shared memory and that shares faulted IPs (Indirect Pointers).

This chapter presents, in detail, the considerations and restrictions involved in the interface between the dynamic environment provided by EPFs and the older, static environment still present in PRIMOS and in existing user-written applications:

- A restriction on the use of static-mode programs by EPFs
- A restriction on the use of static-mode libraries by EPFs
- Static information to avoid in EPFs
- The effect of EPFs on existing shared static-mode applications

RESTRICTION ON THE USE OF STATIC-MODE PROGRAMS BY EPFS

Because the memory used by static-mode programs is all statically allocated, restrictions are placed on their use.

When This Restriction Is Apparent

The effect of this restriction is seen when one static-mode program is suspended, another static-mode program is invoked, and then an attempt is made to continue execution of the first static-mode program. Because both static-mode programs use the same areas of memory, PRIMOS prevents the attempt to continue execution of the first static-mode program after the second static-mode program has executed.

A Sample of This Restriction

For example, ED and SLIST are both static-mode programs. (They may be converted to EPFs in the future, at which point you must substitute two other static-mode programs for the example below.) If a user invokes ED, suspends the program via Control-P, and then invokes SLIST, the act of loading SLIST into memory causes the suspended memory image of ED to be at least partially destroyed.

Because users typically use Control-P to abort a running program rather than to suspend it with the intention of continuing the suspended program later, PRIMOS allows the subsequent invocation of SLIST.

If, after completing the SLIST session, the user attempts to use the START command to continue the suspended ED session, PRIMOS rejects the attempt, as seen here:

```
ED
INPUT
Jill - I am worried that Tom's Tasty Tinsel might not be
able to handle our shipment of 200 stainless steel pizzas.
I've eaten there, and I have noticed their receiving dock
is not really set up for such large items.  Seeing as you're
in charge of all this, I suggest you call them up and
make sure they know what they're getting!  -- Fred

P.S. The order number is                          (user types Control-P)
QUIT.
OK 14:19:48  0.218  0.033  level 2
SLIST FREDDY>TOMS_ORDER#
Fred: the order number for Tom's Tasty Tinsel order of
200 stainless steel pizzas is 0214453.  -- Sue
OK 14:19:53  0.172  0.054  level 2+
RELEASE_LEVEL
Static mode program released. (rls)
OK 14:20:01  0.054  0.000  level 2
START
Attempt to proceed to overwritten program image. (listen_)
ER 14:20:02  0.045  0.000  level 2
```

How to Recover From Encountering the Restriction

If you encounter this restriction, you should release the command stack using the RELEASE_LEVEL -ALL command. In addition, you should use the CLOSE -ALL command to close any file units opened by the restricted invocation.

An alternative method of recovery is to issue the INITIALIZE_COMMAND_ENVIRONMENT command (abbreviated ICE).

A more general solution to this problem is to convert your existing static-mode programs to program EPFs.

Note

These recovery methods recover the user's ability to execute commands up to the full limit of the user's environment. They do not recover the data in the overwritten program. In the example given above, for instance, the edit session is lost.

A General Statement of the Restriction

A generalization of this restriction is that only one static-mode program may be active, whether suspended or running, at any given time for a given process.

Therefore, if the user invokes program A, then quits and invokes program B, then quits and invokes program C, and so on, PRIMOS will automatically terminate active static-mode programs in the sequence (A, B, C, and so on) each time a new static-mode program is invoked. Any subsequent attempt to continue an inactive static-mode program that had been suspended causes PRIMOS to issue an error message.

The Reason Behind the Restriction

As implied by the error message displayed when the restriction is encountered, PRIMOS prevents an attempt to continue a suspended program invocation if it detects that parts of the suspended program image may have been overwritten.

Unlike program EPFs, the placement of procedure and linkage areas for static-mode programs is determined during program load sessions by either the SEG/LOAD utilities or by the programmer during the SEG/LOAD session. Static-mode programs therefore cannot be dynamically placed in memory when they are invoked.

Additionally, most static-mode programs create their own stack bases rather than using the command stack (used by program EPFs). The placement of this stack base is also determined when the program is loaded.

Due to these characteristics, static-mode programs cannot coexist in memory for a particular user process.

RESTRICTION ON THE USE OF STATIC-MODE LIBRARIES BY EPFS

Because the linkage areas for static-mode libraries are all statically allocated, a restriction is placed on their use by program EPFs.

When This Restriction Is Apparent

The effect of this restriction is seen when one program EPF is suspended, another program EPF is invoked, and then an attempt is made to continue execution of the first program EPF. If both the first and second program EPF use the same static-mode library, then PRIMOS prevents the attempt to continue execution of the first program EPF after the second program EPF has executed.

A Sample of This Restriction

In this example, a program EPF named SPOOL_MY_REPORTS, which makes use of the static-mode library SPOOL$, is invoked, suspended via Control-P, and then reinvoked. When the second invocation completes, the user attempts to proceed with the first invocation, and encounters the static-mode library restriction. (The Prime-supplied SPOOL$ library may be converted to a library EPF in the future. At that point, if other static-mode libraries still exist, you must substitute the use of one of them and a program that uses it in the example below to reproduce it. However, it is possible that all static-mode libraries supplied by Prime may be converted to library EPFs in the future, at which point the restriction under discussion is no longer pertinent.)

```
OK, RDY -LONG
OK 10:13:30  0.060  0.000
RESUME SPOOL_MY_REPORTS
GO
Enter start date for report spooling, or <CR> to leave:
—> 08/10/84
Enter end date for report spooling, or <CR> to leave:
—> 08/14/84

Spooling reports for period starting on 08/10/84 and ending on
08/14/84...

08/10/84 is PRT005 (12 records)
08/11/84 is PRT006 (13 records)
08/12/84 is PRT007 (10 records)
08/13/84 is PRT008 (5 records)
08/14/84 is PRT010 (47 records)

Enter start date for report spooling, or <CR> to leave:
—>                                          (user types Control-P)
QUIT,
OK 10:20:46  12.254  9.301  level 2
RESUME SPOOL_MY_REPORTS
GO
Enter start date for report spooling, or <CR> to leave:
—> 08/17/84
Enter end date for report spooling, or <CR> to leave:
—> 08/21/84

Spooling reports for period starting on 08/17/84 and ending on
08/21/84...

08/17/84 is PRT012 (8 records)
08/18/84 is PRT013 (10 records)
08/19/84 is PRT015 (25 records)
08/20/84 is PRT016 (18 records)
08/21/84 is PRT017 (30 records)
```

```
Enter start date for report spooling, or <CR> to leave:
--> (CR)
OK 10:27:31  10.205  7.722  level 2
START  /* User now attempts to continue first invocation.
Attempt to proceed to overwritten program image. (listen_)
ER 10:27:52  0.060  0.000  level 2
```

How to Recover From Encountering the Restriction

If you encounter this restriction, you should release the command stack using the RELEASE_LEVEL -ALL command. In addition, you should use the CLOSE -ALL command to close any file units opened by the restricted invocation.

An alternative method of recovery is to issue the INITIALIZE_COMMAND_ENVIRONMENT command (abbreviated ICE).

The Reason Behind the Restriction

As implied by the error message displayed when the restriction is encountered, PRIMOS prevents an attempt to continue a suspended program invocation if it detects that parts of the suspended program image may have been overwritten.

Although the program in the example is a suspended program EPF, and therefore has not had any of its program image overwritten, the static-mode library SPOOL$, used by the EPF, has had its linkage area overwritten.

When a static-mode library is first invoked by a program EPF, PRIMOS initializes the linkage area for the library. During execution of the static-mode library subroutines, the linkage area for the static-mode library is used and modified by the static-mode subroutines.

If execution of the program EPF is suspended, for example, via Control-P, a subsequent command that runs a program EPF or a static-mode program that also uses the same static-mode library may occur. If this happens, PRIMOS must reinitialize the linkage area for the static-mode library when the library is called, so that the new program will not use subroutines that are operating on undefined data values in the linkage area.

Because the linkage area for a static-mode library is statically located, it cannot be relocated by PRIMOS for the new program that is calling the library.

Therefore, reinitializing the linkage area for that library destroys the previous contents of the linkage area. The previous contents of the linkage area were set during the first invocation of the library by the original program EPF.

PRIMOS detects this condition and prevents the user from continuing under such circumstances so that the original program EPF will not behave in an undefined fashion when it makes subsequent calls to the static-mode library.

STATIC INFORMATION TO AVOID IN EPFS

Whether building a new application as an EPF or converting an existing application to an EPF, you should avoid the use of certain static information by the program. Such static information includes:

- Command line information
- Error information

Static command line information is accessed and manipulated using the subroutines COMANL and RDTK$$; static error information is accessed and manipulated using the subroutines GETERR, PRERR, and ERRSET. An EPF that uses these subroutines may sometimes operate correctly and, at other times, produce invalid results.

Static Command Line Information

The COMANL and RDTK$$ subroutines have been modified at Rev. 19.4 so that static command line information, which prior to Rev. 19.4 was per-process information, is now maintained for each command level. This allows separate programs that use these subroutines to coexist in the same process without disturbing each other. However, if one program that uses COMANL and RDTK$$ directly invokes another program that also uses these subroutines, the second program's use of these subroutines will disrupt the command line information for the first program, and invalid results may be produced. (One program directly invoking another does not produce a change in the user's command level; static command line information is maintained for each command level, but not for each individual program.)

The alternative to using COMANL and RDTK$$ is to design programs so that they accept the command line from the PRIMOS command processor as an argument to the main entrypoint of the program, and so that they parse the command line using a parsing subroutine such as CL$PIX or CMDL$A.

Static Error Information

The GETERR, PRERR, and ERRSET subroutines have not been modified at Rev. 19.4. The static error vector remains a per-process entity. Therefore, two programs that use ERRSET along with GETERR and PRERR to set and retrieve static error information are likely to have adverse effects on each other if they coexist in the same process as EPFs.

The alternative to using these subroutines is to use PRIMOS subroutines that return error codes rather than older versions that set the static error vector; to use the ERRPR$ subroutine to display error messages; and to pass the returned error codes back to the calling subroutines or the PRIMOS command processor via the second argument of the main entrypoint calling sequence.

EFFECT OF EPFS ON EXISTING SHARED APPLICATIONS

In rare cases, existing applications that use shared segments for storage of program code (as opposed to shared data) may be adversely affected by the introduction of library EPFs at Rev. 19.4. In particular, shared static-mode programs and libraries that place faulted IPs (Indirect Pointers) in shared segments will probably exhibit erratic behavior when run on a Rev. 19.4 system. (Faulted IPs are IPs to external subroutines that are satisfied with dynamic links, or DYNTs, during the loading of the application.)

Applications that remain static-mode at Rev. 19.4 should not encounter problems when run under Rev. 19.4 PRIMOS if they do not share faulted IPs. Applications that do share faulted IPs, either explicitly or implicitly (by sharing linkage frames) may be affected at Rev. 19.4 because libraries, which before Rev. 19.4 were statically assigned segments during system coldstart, may at Rev. 19.4 be library EPFs. Therefore, a particular entrypoint in a library EPF may have one address for user A and another address for user B, even though the library EPF is shared by the EPF mechanism. Prior to Rev. 19.4, that entrypoint would always have the same address for all users following a system coldstart.

Shared applications that place some or all of their linkage information in shared memory (segments '2000 through '3777) are likely to encounter this situation, because linkage information typically contains faulted IPs for most subroutines. Applications that explicitly place any faulted IPs in procedure text that is then shared will also encounter this situation.

Normally, all IPs and ECBs are placed in the linkage frame for a procedure; linkage areas for a program are normally placed in per-user (nonshared) memory. Some applications place ECBs in the procedure frame by using the -PBECB option of the compiler or by specifying the ECB pseudo-op of PMA within a PROC block, rather than within a LINK block, in a PMA program. Such applications will encounter no difficulties unless they are converted to EPFs.

Other applications place the linkage frame of one or more procedures in the shared (procedure) segments. Faulted IPs may be present in the linkage frames of such applications. If this is the case, the shared segments are not protected against modification, allowing these faulted IPs to be snapped as the application is used. (A segment-protection value of 700 octal causes the SHARE command to allow all users to modify the segment.) Such applications will likely behave erratically until they are modified so that they do not share linkage frames.

Rarely, applications explicitly place faulted IPs in shared segments. These IPs are explicitly resolved during system coldstart by a program that snaps all of the links for that program before protecting the shared segments against further modification. (Applications that share faulted IPs and explicitly snap them during coldstart are unusual and are not built according to Prime-documented guidelines.) Such applications will probably behave erratically until they are modified so that they do not share faulted IPs.

All of the practices described in the preceding paragraphs are used to reduce the working set of an application. By sharing ECBs, IPs, linkage frames, or all three, less total memory is used when several users run the application.

Effect of Sharing Faulted IPs

An IP that points to the ECB (Entry Control Block) of a procedure starts out as a faulted IP if it points to a dynamically linked object (an entrypoint that is accessed via the dynamic linking mechanism). When an indirect reference occurs through a faulted IP, typically via a PCL IP,* instruction, PRIMOS determines the name of the entrypoint being targeted by the IP. PRIMOS then searches its list of entrypoint names, starting with internal entrypoints, then moving on to items in the user's entrypoint search rule (ENTRY$.SR).

When PRIMOS finds the desired entrypoint, PRIMOS determines the address of the ECB of the target procedure and then replaces the original IP with that address.

As part of finding the desired entrypoint, PRIMOS may map in a new library EPF, assigning it areas of memory in which it is to reside. Therefore, a reference by user A to subroutine PLOTXY through a faulted IP might be resolved by PRIMOS to an ECB at location 4362/1764, whereas the same reference by user B to the same subroutine might be resolved to another copy of the ECB, for the same subroutine, at location 4357/1764.

For IPs placed in per-user (nonshared) memory segments, this poses no problem, because each user has a separate copy of the IP to match the separate copy of the ECB.

However, a program sharing faulted IPs resolves them at system coldstart so that they point to the ECBs of desired entrypoints. Library EPFs containing desired entrypoints are mapped in for the user who first shares the program (typically user 1, which runs the system startup file). Addresses of the desired entrypoints within the library EPFs replace the original faulted IPs as the effective addresses of the ECBs.

Then, when another user invokes the same application later, the same resolved IPs are used (because they reside in shared segments). However, libraries are mapped into memory only as a result of encountering faulted IPs; therefore, the library EPFs referenced by these resolved IPs will probably not be mapped into memory for this second user.

Even if the needed library EPFs are mapped into memory for the user, they may not necessarily reside in the corresponding areas of memory as they did for user 1 when the faulted IPs were snapped.

As a result, the application typically encounters an illegal segment error, an access violation error, or a pointer fault error.

Modifying an Application to Not Share Faulted IPs

To modify an application so that it does not share faulted IPs, you must either change its load sequence so that shared segments are used to contain only procedure code and other constant data, or you must load all of the subroutines it needs into the same application, including those in Prime-supplied libraries.

Modifying the Load Sequence: Modifying the load sequence of an application so that it does not share IPs is the safest approach. It involves changing the load sequence so that only pure code (procedure code) is placed in shared segments and disabling special-purpose programs that snap faulted IPs at coldstart for the application. For example, a CPL program that builds such an application (via SEG) might contain the following line:

```
S/LOAD MODULE1 0 2035 2035
```

The underlined segment number, the second 2035 in the command line, specifies that linkage information (including IPs and the ECB) is to be placed in segment '2035, a shared segment. Modify this line, and lines like it, to place the linkage information in nonshared segments, such as segment '4000. For example:

```
S/LOAD MODULE1 0 2035 4000
```

Then, modify the load sequence for your application so that it performs no processing of the .SEG file, program map, or object files for the purposes of gathering information on the locations of faulted IPs. (Because Prime provides no program for doing this, an example of this cannot be documented here; it is expected that each development group that has built an application that shares IPs has also built its own tools to find faulted IPs.) An application may have no such program, if it leaves shared segments unprotected against user modification.

Finally, find the portion of the system startup file, PRIMOS.COMI (or C_PRMO), that shares the application and modify it so that it no longer runs a program to snap faulted IPs in the shared segment images of the program. If your application has no such program, modify the system startup file to set the protection for shared segments to '600 (read and execute) rather than '700 (read, write, and execute).

Loading In All Subroutines: An alternate solution is to load in all subroutines used by your application that do not reside in PRIMOS itself or in static-mode (Prime-supplied) libraries; that is, load all subroutines used by your application that reside in library EPFs directly into your application in place of the dynamic links it currently loads.

This solution has the disadvantage of increasing the size of your application while duplicating the extra subroutines loaded; other applications will be unable to access the copies of those subroutines loaded into your application, and will instead use the copies in the library EPFs. However, as the size of your application generally affects only the coldstart initialization time of your application, performance should not be reduced; in fact, because the remaining faulted IPs can still be snapped at coldstart, the performance gains realized by constructing your application in this unusual way can be maintained. (Internal PRIMOS subroutines and subroutines in static-mode libraries remain in the same areas of memory for all users after system coldstart.)

However, you must load in the unshared versions of all libraries that your application references. For example, in a particular application that uses the Pascal library, the load sequence might contain:

```
D/LIBRARY PASLIB
D/LIBRARY
```

Replace these statements to load in the unshared versions of the libraries. The default libraries loaded in via a LIBRARY command with no filename are SPLLIB, PFTNLB, and IFTNLB; the unshared versions are NSPLLIB and NPFTNLB. (IFTNLB has no unshared counterpart; it is included in NPFTNLB.) The corresponding statements for the above sample section of a load sequence would therefore read:

```
D/LIBRARY NPASLIB
D/LIBRARY NSPLLIB
D/LIBRARY NPFTNLB
D/LIBRARY
```

Note that the D/LIBRARY command, with no filename, is still given at the end; it results in the loading of DYNTs to static entrypoints (entrypoints into PRIMOS and entrypoints residing in static-mode libraries).

5
Program EPFs

This chapter describes how to design and implement programs as program EPFs.

WHAT IS A PROGRAM EPF?

A program EPF is an executable file system object. A program EPF is generated by you, the programmer, using BIND. It may be used by you, by another user, or by another program.

To a programmer, a program EPF is a file containing a program. To a user, a program EPF is a command. To a program, a program EPF is a subroutine with a predefined calling sequence accessible via one of three PRIMOS subroutines: CP$, EPF$RUN, or EPF$INVK.

The Programmer's View of a Program EPF

Typically, a program EPF contains a completed and working program. A programmer builds a program EPF by following three steps:

1. Entering the program into the system using an editor such as EMACS
2. Compiling the program using a compiler such as F77 or PL1/G, or assembling the program using PMA

3. Binding the program using BIND

Step 1, entering the program into the system, produces source files for the compiler or assembler. Step 2, compiling or assembling the program, uses the source files to produce object files. Step 3, binding the program, uses the object files to produce a program EPF.

The file containing a program EPF has the name program.RUN. The .RUN suffix specifies that the file contains a program EPF. (An alternate suffix for an EPF file, .RPn, is described in Chapter 1.)

This chapter describes how to take full advantage of program EPFs during each of the above steps.

The User's View of a Program EPF

A user runs a program EPF in one of two ways:

- By typing the name of the program EPF as a PRIMOS command
- By using the PRIMOS RESUME command to run the program EPF

General information on entering commands and running programs is provided for users in the Prime User's Guide.

Typically, if you write a program, you provide additional documentation describing the nature and purpose of the program, where it resides, how to run it, and whom to contact if problems arise.

Invoking a Program EPF as a PRIMOS Command: If a program EPF resides in the directory CMDNC0 on the command disk of the system, users invoke the program EPF by simply typing its name as if it were a normal PRIMOS command. The command disk is usually logical disk 0 on a system. For more information, see your System Administrator or the System Administrator's Guide.

Invoking a Program EPF Using the RESUME Command: A user invokes any program EPF by using the PRIMOS RESUME command and specifying the pathname of the program EPF. The user must have Read access to the program EPF.

The Program's View of a Program EPF

A program invokes a program EPF by calling one of several system subroutines, depending on the needs of the invoking program. When one program invokes another program in this manner, the first program is suspended while the second program runs. When the second program finishes, the first program continues running.

The first program may supply input data to the second program when it invokes the second program. It can also accept returned information from the second program when the second program finishes. For example, the first program can receive information from the second program as to whether or not the second program ran successfully.

The ability for one program to call another is not limited to one occurrence; the second program can call a third, and the third can call a fourth. The only limits placed on the invocation of programs from within programs are:

- Resource limits, such as amount of memory and internal table space
- Restrictions placed on the use of static-mode programs (those programs linked using SEG or LOAD rather than BIND)
- Limits set by the System Administrator on the command level breadth

The resource limits are exceeded when PRIMOS is unable to allocate resources to execute another program EPF. A discussion of static-mode program restrictions is found in Chapter 4.

The command level breadth is the number of programs that are active at a given command level. PRIMOS maintains this number; this number is 0 when the user is at PRIMOS command level. When the user runs a program EPF, PRIMOS sets this number to 1. If the program EPF invokes another program EPF, PRIMOS then sets this number to 2. If the second program EPF invokes a third, PRIMOS sets this number to 3, because three programs are active at that command level.

As program EPFs finish and return to their callers, PRIMOS decrements this number. When the original program EPF is reached, PRIMOS sets this number to 1. The original program EPF may then choose to invoke further program EPFs, which causes PRIMOS to again increase the command level breadth. However, when the original program EPF finally finishes, PRIMOS returns this number to 0, and places the user at PRIMOS command level.

The System Administrator has the ability to limit the command level breadth for all users of the system on a per-user basis. Therefore, an attempt by a program EPF to invoke another program EPF may be thwarted by PRIMOS because the maximum limit on the user's command level breadth would be exceeded.

For more information on invoking programs from within programs, see Volume III of this series.

WRITING THE MAIN PROGRAM OF A PROGRAM EPF

In every program EPF, one procedure is defined as the main entrypoint. The main procedure is the procedure that is called by PRIMOS when the program EPF is run. It is identified during the BIND session via the MAIN subcommand. If the MAIN subcommand is not issued, BIND assumes the first procedure linked is the main procedure.

The main procedure of a program EPF should accept no arguments unless command line processing is to occur. See Volume III of this series for information on writing program EPFs that perform command line processing.

Therefore, the main procedure of an F77 program should begin as follows:

```
PROGRAM program-name
```

The main procedure of an FTN program should begin with:

```
SUBROUTINE program-name
```

The main procedure of a PL1/G program should begin with:

```
program-name: PROCEDURE;
```

The main procedure of a PMA program should be structured as described in Chapter 7. However, its ECB should specify that it accepts no arguments, as follows:

```
program-ecb  ECB program-name-start
```

In addition, the END pseudo instruction at the end of the module should specify the ECB that is to serve as the main entrypoint for the module, as follows:

```
END program-ecb
```

If you specify no label following the END pseudo instruction, you must use the MAIN subcommand of BIND when you link the PMA module as the first module of a program EPF, or the program will fail to run.

The MAIN Subcommand of BIND

Use the MAIN subcommand of BIND to specify the main entrypoint of the program EPF. PRIMOS invokes this entrypoint when the EPF is invoked. If the MAIN subcommand is not specified during the linking of a program EPF, BIND defaults to choosing the first ECB linked during the BIND session as the main entrypoint.

The main entrypoint of a program EPF has a predefined calling sequence if it accepts arguments. This calling sequence is described in detail in Volume III of this series.

The DYNT Subcommand of BIND

The DYNT subcommand of BIND controls the production of dynamic links, typically for references to external entrypoints defined in your own personal library. (PRIMOS does not support dynamic linking to common areas.)

You may use the DYNT subcommand to declare specific entrypoint names as dynamic links, using the form:

```
DYNT name-1 [name-2 ...]
```

The DYNT subcommand is useful when no library object (.BIN) file exists to define dynamic links for entrypoints in a library EPF, such as a library EPF you have created for your own personal use.

You should use the DYNT subcommand only for entrypoints in a library EPF that is for your own personal use. Any library supplied to you, or that you supply to other users, should be accompanied by an object file that contains the appropriate DYNTs. This library should be built according to the guidelines shown in Chapter 6 and should be kept in a single, system-wide location. Users of this library should always use the LIBRARY or LOAD subcommand to link the library object file at BIND time; they should not use the DYNT subcommand to produce links to individual entrypoints within the library.

6
Library EPFs

Library EPFs provide a simple and direct way to build and maintain a library of commonly used subroutines for one or more products. Use of library EPFs can be simple, yet flexible enough to meet the demands of different applications. Sophisticated use of library EPFs is possible, but it requires sophisticated knowledge of how your product is organized.

To make the use of library EPFs as simple as possible, PRIMOS provides some intriguing mechanisms that help make the existence of library EPFs transparent to most users. These mechanisms include:

- Library search lists
- Automatic dynamic linking
- On-demand library EPF mapping

As the programmer of a library EPF, you must be aware of how these mechanisms are seen by users, and how to use them during the development process for your product library.

To fully acquaint you with all aspects of library EPFs, this chapter:

- Describes what a library EPF is
- Describes the steps needed to create a library EPF
- Examines the choice of the appropriate type of library EPF in detail

- Explains how to use DBG on a library EPF
- Describes library entrypoint search lists and how to manipulate them
- Describes specific aspects of the EPF mechanism that pertain to library EPFs

WHAT IS A LIBRARY EPF?

A library EPF is an executable file system object. A library EPF is generated by you, the programmer, using BIND. It is used by other programs that invoke it by calling entrypoints in it.

To a programmer, a library EPF is a file containing a program. To a program, a library EPF is a collection of subroutines.

The Programmer's View of a Library EPF

Typically, a library EPF contains a collection of completed and working subroutines program. There are two kinds of library EPF:

- Program-class
- Process-class

The choice as to whether a library EPF should be program-class or process-class is up to the programmer who is building the library EPF. Typically, a library EPF is built as a program-class library EPF; PRIMOS treats a program-class library EPF as part of any program that uses that library EPF. Occasionally, a library EPF is best built as a process-class library EPF; PRIMOS keeps process-class library EPFs separate from program invocations.

A programmer builds a library EPF by following these steps:

1. Enter the subroutines into the system using an editor such as EMACS or ED.
2. Compile the subroutines entered in Step 1 using a compiler such as F77 or PL1/G, or assemble the subroutines using PMA.
3. Determine how the library is to be organized.
4. Determine which subroutines in the library are to be considered entrypoints into the library, and ensure that there are no naming conflicts between your entrypoints and entrypoints in other libraries.

5. Link the subroutines compiled in Step 2 into one or two library EPFs using BIND, indicating which subroutines are entrypoints.

6. Build a PMA file listing entrypoints as dynamic links (DYNTs).

7. Assemble the PMA file built in Step 6, generating an object (.BIN) file containing DYNTs to your library.

8. Use EDB to make the .BIN file generated in Step 7 a non-force-linked library.

9. Install the .BIN file built in Step 8 in the appropriate directory.

10. Install the library EPF built in Step 5 in the appropriate directory.

11. Modify the appropriate entrypoint search list to reference the library EPF installed in Step 10.

Step 1, entering the subroutines into the system, produces source files for the compiler or assembler. Step 2, compiling or assembling the subroutines, uses the source files to produce object files. Step 3, determining how the library is to be organized, requires you to assess the way in which individual subroutines in your library need to be initialized when invoked. Step 4, determining the entrypoints for your library, requires you to check your entrypoint names against a Prime-supplied list of reserved entrypoint names (provided later in this chapter) and against other library EPFs, not provided by Prime, used at your installation.

Step 5, linking the subroutines using BIND, uses the object files to produce a library EPF. The file containing a library EPF has the name library.RUN. The .RUN suffix specifies that the file contains an EPF. (An alternate suffix for an EPF file, .RPn, is described in Chapter 1.)

Steps 6 through 9 provide the file to be loaded or linked by programs that are to use your library. They use the LOAD or LIBRARY subcommand of either SEG or BIND, specifying the .BIN file installed in Step 9, to resolve references to subroutines in your library to dynamic links (DYNTs).

Steps 10 and 11 make your library EPF available to whoever uses an entrypoint search list that specifies your library EPF. Running programs that encounter dynamic links specifying subroutines that are entrypoints in your library EPF are automatically connected to your library EPF. Such programs acquire this ability by loading or linking the library (.BIN) file installed in Step 9 if the entrypoint search list names your library EPF.

This chapter describes how to take full advantage of library EPFs during each of the above steps.

The Program's View of a Library EPF

To a program, a library EPF is invoked by calling a subroutine that is an entrypoint to that library EPF. Unlike the main entrypoint of a program EPF, PRIMOS imposes no constraints upon the calling sequence of the subroutine.

STEPS IN BUILDING A LIBRARY EPF

This section describes the steps performed in building a library EPF. Some of the steps require even further detailed explanation for certain cases; subsequent sections in this chapter address these needs.

It is important that you read the descriptions of all of the steps before you begin designing and coding your library EPF. In particular, the entrypoint-naming issues described in Step 4 may impact your design specifications.

Step 1 - Enter the Subroutines

You enter the subroutines for a library EPF just as you would for a program EPF, by using a text editor such as EMACS or ED. No subroutines should be coded as PROGRAM modules; they should all be SUBROUTINE, PROCEDURE, or FUNCTION modules. No restrictions are placed on their calling sequences by PRIMOS.

Step 2 - Compile the Subroutines

You compile or assemble the subroutines for a library EPF using one of Prime's compilers or PMA. The compiler must generate 64V-mode or 32I-mode code; a PMA program must contain the pseudo-op SEG or SEGR. These requirements are described in Chapter 7.

Step 3 - Determine How the Library Is to be Organized

You now consider whether the library should be a single program-class library EPF, a single process-class library EPF, or one of each. Typically, program-class library EPFs are the best choice. Occasionally, however, it improves performance to put subroutines in a process-class library EPF, as long as they will perform correctly. See the section below, entitled CHOOSING THE RIGHT TYPE OF LIBRARY EPF, for detailed information on making the decision between building a single program-class library EPF, a single process-class library EPF, or two library EPFs (one of each type).

The simplest, and often the most appropriate, decision is to build one program-class library EPF. Only in rare cases does this approach not produce a working library. Advantages of process-class library EPFs are primarily in terms of performance.

Step 4 - Determine Library Entrypoints

You now build a list of the subroutines that are to be considered entrypoints in your library. You will use this list for two purposes:

- To identify entrypoints in the library EPF during the BIND session
- To identify external entrypoint references as dynamic links in programs that use your library EPF

Subroutines that are not made entrypoints to a library EPF cannot be called by subroutines outside that library EPF, unless their addresses are provided by other entrypoints in the library EPF via ENTRY VARIABLE (or similar) functionality.

Prime reserves many names for its own use as entrypoint names. These names are listed next.

Reserved Names

The complete list of entrypoint names reserved by PRIMOS is on the following page. In addition to the names in this list, names containing a $ symbol are reserved by PRIMOS and Prime-supplied libraries.

> WARNING
>
> Do not attempt to use any of the entrypoints listed unless other documentation specifically explains it. Using undocumented PRIMOS subroutine entrypoints may result in unusual behavior by PRIMOS subsystems. In addition, such subroutines may be removed or changed at any point without warning.

Note

Prime guarantees that no new names will be added to the following list. In other words, no reserved names will be added to this list; the only changes made to this list will be the removal of names that are used only internally by Prime products as the names are changed to have $ symbols in them.

ACKRCT
ADD_QREC
ADOCRD
ADQREC
ADRESS
AD_CMD
ALLOC
APPEND
APROTO
ATLIST
ATMAIN
ATTDEV
AVAIL
BCKUPB
BSCMAN
C1IN
CALFC_
CFI
CIRLOG
CKINST
CKNDNM
CLEARS
CLREAD
CLRLIN
CLR_FLDS
CLSDOC
CMD_POST
CMD_PRE_
CNIN
COMANL
CONTRL
CRAWL_
CREUFD
DATE_A
DCTEXS
DECR_HOP
DEFILE
DELAY
DELAY_
DELETE
DELETE_Q
DELOAS
DH3270
DIDNUM
DIRSER
DISPLA
DMIDAS
DMLCP
DNUMID
DOSSUB
DPTINI
DPTOFF
DRAIN_QU
EM3270
ENCRYP
EPF_ERR
EPF_RL
ERASE
ERROPN
ERRRTN
ERRSET
EVAL_A
EXIT
EXTRAC
FILERR
FILHER
FINDPG
FIND_U
FIND_UID
FNDREC
FNDWRD
FORCEW
FREE_DES
FREE_Q_R
FSCHOC
GALLKS
GCHAR
GETADR
GETENT
GETERR
GETINPG
GETINYB
GETREG
GETSLT
GET_REPL
GFILKS
GINFO
GORDNC
GOREAD
GTDOCR
GTWORD
GUSLKS
HASH_U
HASH_UID
ICMTB_
ICPL_
ICS2CT
ID
INCPTR
INITP1
INIT_NPX
INIT_Q_S
INTCM_
IQNET
IQUSER
ISFEPF
JUSTRT
LCKGRP
LIBTBL
LIST_SRL
LNGCMP
LOCK
LOG_EVEN
LOG_RECO
MCSDAT
MCSTOD
MOVB
MOVE
MOVEB
MOVWDS
MSGCTL
NETCHK
NETFIG
NETPRC
NETSET
NEWS
NOTDST
NPXPRC
NXTLIN
OAUSER
OERRTN
OPNDFL
OPNDOC
OPNQFL
OWL2
P1IB
P1IN
P1OB
P1OU
P2UPCS
PACK_BIT
PACK_CHA
PACK_INT
PARS_ATT
PARTCL
PASSWD
PCREAT
PEXIT
PFIL2A
PFLM9E
PHDBG
PINIT
PINLNK
PK2LDV
PRIBLD
PRICON
PRVSB_
PTRAP
PUTBL
PUTSLT
PUT_HOP
QPARSE
QPOST
QUITHD
QUOTE_
R0BASE
R3FALT
RDASC
RDBIN
RDNPAG
RDPRCN
RECYCL
RELGRP
REMANS
REMUSR
REPOST
RESTART_
RGSTRY
RJDBG
RJMNIT
RJPROC
RMLOCK
RPTSPL
RQUEST
RSTBL
SAL_HP
SCANB
SCHAR
SEARCH_C
SEARCH_H
SECBLD
SEGCON
SELANG
SETATT
SETNAM
SETREG
SET_SRL
SET_VERS
SFR_CFSC
SFR_HP
SHRLIB
SH_CMD
SLAVE
SLAVER
SOUR3_
SPLCHK
SRWREC
STK_EX
STPNC
STRBL
STRTPH
STUFF
SUBMIT
SWFBK_
SWFIM_
SWINTQ
T1IB
T1IN
T1OB
T1OU
TBLRED
TIDEC
TIHEX
TIMDAT
TIMREC
TIMSLT
TIOCT
TM3270
TMDISP
TNOU
TNOUA
TODEC
TOHEX
TONL
TOOCT
TRNRCV
TRTYPE
TRVERSIO
TSATRC
TSTAMP
UDTDRY
UDTF07
UNCAN_ME
UNLOCK
UNPACK_A
UPCASE
UPDATE_S
USERID
USRPRM
VMMSG
VMMSG2
VMMSG3
VREMID
WHATIT
WRASC
WRBIN
WRITLINE
WRTPG
XLACPT
XLASGN
XLCLR
XLCLRA
XLCONN
XLGCON
XLGVVC
XLUASN
XMTRCV

Step 5 - Linking the Subroutines

You now link the subroutines using BIND to create a library EPF of the appropriate type, or to create two library EPFs, one of each type.

For a program-class library EPF, the link sequence is:

```
BIND library-EPF-filename
LIBMODE -PROGRAM
LOAD module-1
LOAD module-2
      .
      .
      .
ENTRYNAME name-1 [name-2 ...]
LIBRARY [special-library-1 ...]  /* if needed
LIBRARY
RESOLVE_DEFERRED_COMMON
MAP [map-filename] [options]
FILE
```

For a process-class library EPF, the link sequence is:

```
BIND library-EPF-filename
LIBMODE -PROCESS
LOAD module-1
LOAD module-2
      .
      .
      .
ENTRYNAME name-1 [name-2 ...]
LIBRARY [special-library-1 ...]  /* if needed
LIBRARY PROCESS_CLASS
LIBRARY
RESOLVE_DEFERRED_COMMON
MAP [map-filename] [options]
FILE
```

The differences between linking a program-class library EPF and a process-class library EPF are:

- Use of the LIBMODE -PROCESS subcommand rather than LIBMODE -PROGRAM
- Use of the LIBRARY PROCESS_CLASS subcommand immediately before the LIBRARY subcommand

The LIBMODE subcommand specifies the type of library EPF to be generated. The LIBRARY PROCESS_CLASS subcommand links a library that causes dynamic allocation performed by your library EPF to be done from process-class, rather than program-class, memory.

Although not required to link a library EPF, the RESOLVE_DEFERRED_COMMON and MAP subcommands are recommended for use when debugging a library EPF. See also the section below entitled HOW TO USE DBG ON A LIBRARY EPF.

An Easy Way to Declare Entrypoints: An easier way to declare entrypoints to your library EPF is to use the ENTRYNAME -ALL and ENTRYNAME -NONE subcommands. Between use of these two subcommands, BIND automatically makes any subroutines linked via the LOAD or LIBRARY subcommands into entrypoints for the library.

Therefore, the template you would use for the appropriate section of the above two build file templates is:

```
ENTRYNAME -ALL
LOAD entrypoint-module-1
LOAD entrypoint-module-2
      .
      .
      .
ENTRYNAME -NONE
LOAD other-module-1  /* if needed
LOAD other-module-2
      .
      .
      .
LIBRARY [special-library-1 ...]  /* if needed
```

Make certain that you issue an ENTRYNAME -NONE subcommand before using the LIBRARY subcommand. Otherwise, you are likely to produce a library EPF that either will not execute correctly or that has entrypoint names that conflict with Prime-supplied libraries.

Step 6 - Building a PMA Entrypoint File

You now build a single PMA file that declares all of the entrypoints to your library EPF (or both library EPFs) as dynamic links. This file has the format:

```
* List of dynamic links for MYLIBRARY.RUN.
          SEG
          SYML
          DYNT entrypoint-1
```

```
END
SEG
SYML
DYNT entrypoint-2
END
 .
 .
 .
```

An easy way to build this file is to enter the entrypoint names into a file named ENTRYPOINTS, one name per line. For example:

```
INIT_LINE
GET_CHAR
NEW_CHAR_FOR_LINE
CLEAR_LINE
BACKSPACE_LINE
END_LINE
```

Now, enter and run the following CPL file to build a file named ENTRYPOINTS.PMA:

```
&DATA ED

      LOAD ENTRYPOINTS
      TOP
      N;IB          SEG : SYML;N;GM I/          DYNT /FI/ : END/;*
      FILE ENTRYPOINTS.PMA
      &END
```

This produces the following ENTRYPOINTS.PMA file when run on the ENTRYPOINTS file built in the earlier example:

```
SEG : SYML
DYNT INIT_LINE : END
SEG : SYML
DYNT GET_CHAR : END
SEG : SYML
DYNT NEW_CHAR_FOR_LINE : END
SEG : SYML
DYNT CLEAR_LINE : END
SEG : SYML
DYNT BACKSPACE_LINE : END
SEG : SYML
DYNT END_LINE : END
```

Step 7 - Assemble the Entrypoints File

You now assemble the entrypoints file by issuing the command:

```
PMA ENTRYPOINTS -LISTING NO
```

This produces a file named ENTRYPOINTS.BIN. (You usually do not need to produce a listing of the file. If you wish to produce a listing, omit the -LISTING NO specification.) For example:

```
OK, PMA ENTRYPOINTS -LISTING NO
0000 ERRORS (PMA Rev. 19.4)
OK,
```

Step 8 - Use EDB to Generate Library File

You now use EDB, the Binary Editor, to generate a new version of the ENTRYPOINTS.BIN file that does not forcibly load or link itself into whatever program is using it. A CPL program to perform this step is:

```
&DATA EDB ENTRYPOINTS.BIN MYLIBRARY.BIN
      RFL
      COPY ALL
      SFL
      QUIT
      &END
```

You now have a file named MYLIBRARY.BIN that can be installed as the library file that programs can load or link to use your library EPF.

For example, if you named the CPL file shown above FIX_LIB.CPL, and ran it on the ENTRYPOINTS.BIN file produced in the earlier sample PMA invocation, the following output would result:

```
OK, RESUME FIX_LIB
[EDB rev 19.4]
ENTER,        RFL
ENTER,        COPY ALL
INIT_LINE                          GET_CHAR
NEW_CHAR_FOR_LINE                  CLEAR_LINE
BACKSPACE_LINE                     END_LINE

.BOTTOM.
ENTER,        SFL
ENTER,        QUIT
OK,
```

Notice how the entrypoint names are listed, two per line.

See Chapter 10 for more information on EDB.

Step 9 - Install the Library File

You now install the library file, named MYLIBRARY.BIN in the above example, into the appropriate directory. For system-wide libraries, the LIB UFD is appropriate. For example:

```
COPY MYLIBRARY.BIN LIB>MYLIBRARY.BIN -NO_QUERY -DTM -REPORT
```

When a library is installed in UFD LIB, programs can load (SEG) or link (BIND) it by issuing the SEG or BIND subcommand:

```
LIBRARY library-filename
```

In the example used, the LIBRARY MYLIBRARY subcommand would be used.

Alternatively, you may wish to place the library file in a directory common to users in your project. In this case, programs must specify the full pathname of the library file when loading or linking it. They may use either the LOAD or LIBRARY subcommand of SEG or BIND with a full pathname, although the LIBRARY subcommand is recommended because its name communicates more clearly to someone reading the program's build file what the purpose of the file is.

Step 10 - Install the Library EPF

You now install the library EPF file built in Step 5. As with Step 9, you may install the library EPF in either a system-wide directory or in a directory common to users who are to use it. The system-wide library directory for library EPFs is the LIBRARIES* UFD. For example:

```
COPY MYLIBRARY.RUN LIBRARIES*>MYLIBRARY.RUN -NO_QUERY -DTM -REPORT
```

Whether installed in the system-wide LIBRARIES* UFD or in another directory, the library EPF is not usable until the next step, when its full pathname is added to the entrypoint search list of users that are to make use of the library EPF or of programs that use the library EPF.

Step 11 - Modify the Entrypoint Search List

You now modify the appropriate entrypoint search list so that the library EPF you installed in Step 10 is accessible by the appropriate users.

If you have installed the library EPF in the system-wide LIBRARIES* UFD, then you typically modify the system-wide default entrypoint search list, SYSTEM>ENTRY$.SR. The following sample session inserts the search rule LIBRARIES*>MYLIBRARY.RUN at the bottom of the SYSTEM>ENTRY$.SR search list:

```
OK, ED SYSTEM>ENTRY$.SR
EDIT
BOTTOM
INSERT LIBRARIES*>MYLIBRARY.RUN
FILE
SYSTEM>ENTRY$.SR
OK,
```

Caution

Typically, you do not have access to SYSTEM>ENTRY$.SR unless you are the System Administrator. If you modify it, it is possible that you might unknowingly render it unusable, such as by putting one search rule in twice (a duplicate rule). If this happens, not only will users be affected, but a subsequent coldstart of the system may render the supervisor terminal nearly ineffective. In such a situation, you will be unable to use editor ED to fix the file, as ED references faulted IPs to call system subroutines via the dynamic linking mechanism.

The solution to this problem is to use the nonshared editor, NSED, rather than the shared editor ED, to fix the default search list file. NSED runs under PRIMOS II, and therefore does not ever reference faulted IPs.

Alternatively, you may modify the entrypoint search list of the users who are to use the library EPF, or ask them to make the modifications themselves. If these users are using the system-wide default entrypoint search list, then you must construct an entrypoint search list for them that includes both the system-wide default search list and your own library EPF.

For example, if the users all have access to a directory named PROJECT_A, in which you have already installed the library EPF (and possibly the library file named MYLIBRARY.BIN), you might type:

```
OK, ED
INPUT
-SYSTEM
PROJECT_A>MYLIBRARY.RUN
(CR)
EDIT
FILE PROJECT_A>ENTRY$.SR
OK,
```

After you do this, all users who are to use this library must place a SET_SEARCH_RULES command in their LOGIN.CPL or LOGIN.COMI file, as follows:

```
SET_SEARCH_RULES PROJECT_A>ENTRY$
```

Note

Whether you modify the system-wide default entrypoint search list SYSTEM>ENTRY$.SR, modify some other entrypoint search list, or create a new search list, users who are already logged in must issue the SET_SEARCH_RULES command before they can run a program that uses your library EPF. (A user who has the appropriate SET_SEARCH_RULES command in his or her LOGIN.CPL file may simply issue the INITIALIZE_COMMAND_ENVIRONMENT command, as may all users if you have modified the system-wide default entrypoint search list.) If a user complains that a LINKAGE_FAULT$ condition was signaled indicating a failure to link to an entrypoint in your library EPF, it may be that the user is not using an entrypoint search list that includes your library EPF. Ask the user to issue the LIST_SEARCH_RULES command (abbreviated LSR) and ensure that your library EPF is listed therein.

If the user has the correct entrypoint search list, then use the LIST_LIBRARY_ENTRYPOINTS command (abbreviated LLENT) to ensure that the desired subroutine is in fact an entrypoint in your library EPF. See the section below, entitled EXAMINING ENTRYPOINT LISTS, for more information on this command.

CHOOSING THE RIGHT TYPE OF LIBRARY EPF

A library EPF must be either a program-class or process-class library EPF, as described earlier in this Chapter. This section explains how you determine the appropriate classification for subroutines in your library EPF.

The decision as to whether a library EPF is process-class or program-class is actually made on a per-subroutine basis, and includes such factors as:

- How the subroutine uses its linkage area
- How procedures external to the subroutine are classified

In general, the simplest decision is to put all of your subroutines into a program-class library EPF. In most cases, this will produce a working library EPF, although the performance of the library EPF may not be as good as if a process-class library EPF were used.

Performance for a process-class library EPF is often better than that of the same subroutines collected as a program-class library EPF because the linkage area for the library EPF need not be reallocated and reinitialized each time a program using the library EPF is run.

However, subroutines in a process-class library EPF must observe certain restrictions on their use of linkage areas and other external procedures. These restrictions are:

- A subroutine in a process-class library EPF may not call a procedure in a program-class library EPF or a static-mode library.
- Because of the above restriction, a subroutine in a process-class library EPF may not perform any language-directed I/O operations. (No PRIMOS-resident subroutine ever performs language-directed I/O operations; rather, the subroutines that are called upon to perform language-directed I/O operations call PRIMOS-resident subroutines to accomplish their tasks.)
- A subroutine in a process-class library EPF should not modify any data in its linkage area, except in certain special cases.

Both restrictions can be difficult to check for in a given subroutine. PRIMOS does detect and prevent a violation of the restriction against a process-class library EPF subroutine invoking a program-class library EPF or static-mode procedure. However, PRIMOS cannot detect a potentially invalid use of data in the linkage area.

The remainder of this section describes the steps used to determine whether you want one library EPF of a particular class, or two library EPFs (one of each class).

In summary, these steps are:

1. Determine the class requirements of each subroutine in your library.

2. Determine the class requirements of your library EPF using the subroutine requirements data.

If your library EPF must meet performance constraints, then the first requirement can become somewhat complicated. Process-class subroutines tend to have better performance than program-class subroutines because they tend to require complete initialization of their linkage areas less often, but they must meet more stringent requirements (such as not being able to call a program-class subroutine).

First, you determine which subroutines must be program-class subroutines based on two absolute rules listed below.

Second, you examine the remaining subroutines, and determine which of those should be program-class subroutines based on their use of static data.

Third, you consider specific cases where the usage of static data by a subroutine implies that it must be in the program class, but in fact the nature of the static data it uses allows it to be in the process class.

Finally, you consider specific cases where the usage of static data by a subroutine indicates a need to redesign the subroutine (and probably its external interface) so that it can be in the process class.

Determining the Class Requirements of a Subroutine

It is most desirable for a subroutine to work properly as a process-class subroutine. Process-class subroutines do not incur the overhead of allocating and initializing their linkage areas each time they are invoked by a new program.

Instead, their linkage areas are allocated and initialized only the first time the process-class library EPF to which they belong is mapped into memory, and remain valid until the same process-class library EPF is unmapped from memory. If a program that uses that process-class library EPF is run several times between the mapping of the library EPF and its unmapping, the linkage will still be allocated and initialized only once.

However, due to the restrictions described above, not all subroutines will work when built into a process-class library EPF. For the most part, all subroutines will work when built into program-class library EPFs, which do incur linkage allocation and initialization overhead for each program invocation.

The following sections are designed to help you determine, for each subroutine, whether it must be a program-class subroutine or can be in either class.

This section is split up into several rules. Some of these rules are absolute rules that cannot be violated. Other rules apply to most subroutines, but exceptional cases are listed or described.

Subsequent sections discuss ways in which a subroutine can be examined in greater detail to determine if it is, or can be made, a process-class subroutine.

Rule 1 - Restriction on Library Class Mixing: It is an absolute rule that a process-class library EPF subroutine cannot call a subroutine within a program-class library EPF or a static-mode library. If this is attempted, PRIMOS will produce an error message similar to the following:

```
Error: condition "LINKAGE_ERROR$" raised at 4342(3)/12506.
Attempt to link to program class library EPF entrypoint "ATIDEV"
from a process class EPF.
ER!
```

It is possible for a process-class subroutine to call some other subroutine that then calls a program-class or static-mode library subroutine. However, in most cases, this cannot be done because this rule must be reapplied.

Specifically, if process-class subroutine A calls subroutine B which calls program-class (or static-mode library) subroutine C, then subroutine B cannot be a process-class subroutine, or Rule 1 would be violated. In addition, subroutine B cannot be a program-class or static-mode subroutine, or again, Rule 1 would be violated when subroutine A calls subroutine B.

This situation will be valid only if subroutine B is part of a program EPF or a static-mode program. PRIMOS will treat the invocation of subroutine C by subroutine B as being a program-to-program invocation, and will allow it, since PRIMOS will be able to properly allocate and initialize the linkage area for the program-class library EPF or static-mode library containing subroutine C.

However, subroutine A cannot call subroutine B if it is not part of a library unless the entrypoint for subroutine B is passed to subroutine A as part of its calling sequence or through a common area. (For example, a PL1/G ENTRY VARIABLE declaration provides this functionality.)

Therefore, under most circumstances, once a process-class library subroutine is invoked, only process-class subroutines or PRIMOS direct entry subroutines can be called until the process-class library

subroutine returns to its caller. Exceptions to this statement occur only when non-library subroutine calls occur during this period. (Non-library subroutine calls can also occur as a result of a condition being signaled. See the Subroutines Reference Guide for information on writing condition handlers.)

Rule 2 - Restriction on Use of Language I/O: If a subroutine makes use of language-directed I/O, it must be made a program-class subroutine. Language-directed I/O includes statements such as READ, WRITE, ENCODE, DECODE, and OPEN in FORTRAN, PUT, GET, OPEN, CLOSE in PL1/G.

This rule is, in fact, a corollary to Rule 1. All Prime-supplied languages generate subroutine calls to perform language-directed I/O. All such languages provide their runtime support of language I/O as program-class library EPFs. To permit correct management of data buffers between program invocations, language I/O library EPFs must be program-class.

Rule 3 - Problem When Storing Data in Linkage Areas: A subroutine that uses its linkage area (static storage) to store data will probably not function correctly if built into a process-class library EPF.

Exceptions occur for faulted IPs that are resolved by the PRIMOS dynamic linking mechanism and for imaginary IPs that are converted to actual IPs by the PRIMOS EPF invocation mechanism. Such IPs are automatically generated by all Prime-supplied languages, including PMA. However, PMA programmers may explicitly specify an IP in their linkage text that is covered by this exception only if the subroutine never attempts to modify the contents of the IP.

Other special-case subroutines, such as random number generators, may be considered exceptions to this rule. This is discussed later in this chapter.

Determining the Use of Static Data by a Subroutine

If your product does not have stringent performance criteria to meet, it is recommended that you not devote time attempting to determine which subroutines are process-class and program-class. At this point, if you don't know whether some of your subroutines require placement in a program-class library EPF, you should assume that they do.

However, if your product does have performance standards to meet, it may be worthwhile to invest the time needed to determine precisely which subroutines can and cannot be safely made process-class subroutines. It is even conceivable that redesigning the internal operation (and perhaps external interface) of a few chosen subroutines so that they may execute as process-class subroutines would be a reasonable investment of your time, if the resulting performance increase justifies it.

The key issue that determines whether a subroutine must run as a program-class subroutine involves its use of data in the linkage area. This includes:

- Static data (declared as STATIC in PL1/G, with DATA statement in FORTRAN, and, in PMA, via LINK pseudo instruction followed by DATA, OCT, DEC, BSS, BSZ, ECB, IP, and so on)

- Common data (declared as STATIC EXTERNAL in PL1/G, with COMMON statement in FORTRAN, via COMM and EXT pseudo instructions in PMA)

The reason the linkage area is the crux of the issue is that the linkage area is not reallocated and reinitialized for a process-class library EPF when a new program calls a member subroutine. Therefore, if any of the data in the linkage area for the library is program-related, the execution of the second program that calls the process-class library EPF after it is mapped into memory may produce inaccurate results or cause error conditions. See Chapter 5 for more information on such restrictions.

If a subroutine stores data in the linkage area, and it uses that data at any other point, then the subroutine is probably not reentrant. A non-reentrant subroutine must typically be put in a program-class library EPF, to prevent it from misbehaving after multiple invocations by separate programs.

Here is a sample PL1/G subroutine that is not reentrant:

```
average: proc(number) returns(fixed bin(15));

dcl number fixed bin(15); /* The newest number. */

dcl count fixed bin(15) static init(0), /* # of numbers. */
    total fixed bin(31) static init(0); /* Total value. */

count=count+1; /* Another number. */
total=total+number; /* Total it up. */

return(divide(total,count,15)); /* Return quotient of average. */
end; /* average: proc */
```

This subroutine is meant to be called in the following way:

```
do_average: proc;

dcl current_avg fixed bin(15),
    next_number fixed bin(15);

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

call tnou('Enter numbers.  Type 0 to stop.',31);
current_avg=0;
next_number=-1;

do while(next_number^=0);
   call tnoua('Enter next number: ',19);
   call tidec(next_number);
   if next_number^=0 then current_avg=average(next_number);
   end;

call tnoua('The average is ',15);
call tovfd$(current_avg);
call tnou('',0);

end;
```

The AVERAGE subroutine uses STATIC INIT for its averaging data, so that the data values are maintained between calls to the AVERAGE subroutine. It modifies the STATIC INIT storage during execution. This makes it nonreentrant even within a given program. That is, even within one program, the AVERAGE subroutine can be used to calculate the average of only one stream of numbers at a time.

In fact, as it currently exists, it can handle only one stream of numbers for one entire program execution, because there is no method to reinitialize the STATIC INIT data. Even if an alternate entrypoint existed to do this, the subroutine would still be able to manage only one stream of numbers at a time.

The single-stream restriction on the AVERAGE subroutine is not a problem if the calling program needs to calculate an average for only a single stream of numbers at a time. However, it does require that if AVERAGE is made part of a library EPF, it must be a program-class library EPF. That way, separate linkage is allocated and initialized for each different program that uses the AVERAGE subroutine.

If program A uses AVERAGE, and program B is then run and it also uses AVERAGE, the fact that AVERAGE is in a program-class library will prevent program B from simply continuing the calculation of average

number values established in program A. Instead, program B will start with a fresh copy of the averaging data.

When Nonreentrant Subroutines Should Be Process-class

There are certain cases where a nonreentrant subroutine, as determined by its use of static storage, should actually be in a process-class library EPF. Generally, a subroutine that uses static data to store and use only process-related information, rather than program-related information, may be a nonreentrant subroutine that can be installed in a process-class library EPF.

Process-related information includes such data as: username, user number, user's terminal type, the name of the system, today's date, limits on the user's use of command level depth and breadth and on the number of static and dynamic segments, user's project id, and so on. This information is generally process-related or system-related, rather than program-related. Typically, it does not change during the life of a process, and hence does not need to be reinitialized each time a new program calls a library EPF subroutine that maintains this type of information.

However, carefully consider whether the use of a process-related datum is in itself process-related or program-related. For example, today's date is usually process-related, but it may be important for a program to acquire an up-to-date value, instead of a value that may have been put into the linkage area some time ago, including (possibly) yesterday.

For a more tangible example of a subroutine that is nonreentrant, but uses only process-related data in the linkage area, consider the following PL1/G subroutine:

```
get_username: proc returns(char(32));

dcl 1 timdat_info static,
      2 date char(6),
      2 time fixed bin(15),
      2 ticks fixed bin(15),
      2 meters (4) fixed bin(15),
      2 tps fixed bin(15),
      2 user_number fixed bin(15),
      2 user_name char(32);

dcl have_info bit(1) static init('0'b);

dcl timdat entry(1,2 char(6),2 fixed bin(15),2 fixed bin(15),
         2 (4) fixed bin(15),2 fixed bin(15),2 fixed bin(15),
         2 char(32),fixed bin(15));
```

```
if ^have_info
   then do;
        call timdat(timdat_info,28);
        have_info='1'b;
        end;

return(user_name);
end;
```

This subroutine returns the username of the current user. It incurs the added overhead of calling the system TIMDAT subroutine only when it is first invoked.

Even though GET_USERNAME uses static storage in a nonreentrant fashion, you can see that because all of its static storage identifies process-wide data (the user name), it can be in a process-class library EPF. It will be more efficient there than in a program-class library EPF.

Converting a Nonreentrant Subroutine to be Reentrant

If the performance of a particular group of program-class subroutines needs to be increased, it is possible that converting them to process-class subroutines will help result in the needed performance improvements.

To do this, you must convert the subroutine or group of subroutines to a reentrant entity. This often requires major internal changes, probably a rewrite of the target modules, possibly even rewriting into a different language. For example, PL1/G handles the smooth construction of reentrant subroutines quite well, due to its ability to handle pointer manipulation and based structure declaration.

If PL1/G cannot be used, PMA is an alternative. Here, the XB register is often substituted for references to static data in the linkage area that were once LB relative.

In any case, such a conversion often requires changes in the external appearance of the target modules. All uses of the target modules may have to be changed to accommodate the new calling sequences of the target modules. For this reason, it is recommended that you design new interfaces to allow full reentrancy, both inside and outside a single program.

Simple Conversion: To understand how to convert a target module, we'll look at the AVERAGE subroutine, shown earlier. A simple conversion would be to move the small amount of static data out of the linkage area by making it part of the calling sequence of the subroutine. Initialization of this data would then be left to the calling program.

The resulting version of AVERAGE would appear thus:

```
average: proc(number,count,total) returns(fixed bin(15));

dcl number fixed bin(15),  /* The newest number. */
    count fixed bin(15),  /* # of numbers. */
    total fixed bin(31);  /* Total value. */

count=count+1;  /* Another number. */
total=total+number;  /* Total it up. */

return(divide(total,count,15));  /* Return quotient of average. */
end;  /* average: proc */
```

Because the calling sequence of the subroutine has been changed, the method of calling the subroutine would have to be adjusted for all users of the subroutine, as indicated in the following example:

```
do_average: proc;

dcl current_avg fixed bin(15),
    avg_number fixed bin(15),
    avg_total fixed bin(31),
    next_number fixed bin(15);

dcl tnou entry(char(80),fixed bin(15)),
    tidec entry(fixed bin(15)),
    tovfd$ entry(fixed bin(15)),
    tnoua entry(char(80),fixed bin(15)),
    average entry(fixed bin(15),fixed bin(15),fixed bin(31))
         returns(fixed bin(15));

call tnou('Enter numbers.  Type 0 to stop.',31);
current_avg=0;
avg_number=0;
avg_total=0;
next_number=-1;

do while(next_number^=0);
   call tnoua('Enter next number: ',19);
   call tidec(next_number);
   if next_number^=0 then current_avg=average(next_number,
        avg_number,avg_total);
   end;

call tnoua('The average is ',15);
call tovfd$(current_avg);
call tnou('',0);

end;
```

As a result of these changes, the AVERAGE subroutine can become a process-class subroutine. In addition, the calling program can use it for averaging one stream of numbers at a time. The calling program may use several copies of AVG_NUMBER and AVG_TOTAL to keep the number streams separate.

However, this is a limited form of coping with the nonreentrancy problem for two reasons:

- It requires all calling programs to perform the initialization that is best performed by the target module
- To replace large amounts of static data, long (expensive) calling sequences would be needed

A General Approach to Conversion: A more general approach is to separate out the AVERAGE subroutine into three separate procedures. One procedure, INIT_AVERAGE, allocates storage for and initializes the data for a specific number stream. The second procedure, DO_AVERAGE, actually performs the computation. The third procedure, END_AVERAGE, is called to indicate the end of the calling program's need for the maintenance of data on a particular number stream, and hence deallocates the storage for that stream.

To make use of the efficient way in which the Prime 50 Series machines manipulate pointers, the identifier for a number stream will be a pointer. The INIT_AVERAGE, when invoked, returns a pointer to be used to identify that particular number stream to DO_AVERAGE and END_AVERAGE. The same pointer identifies the area in memory in which the number stream data is stored, for use by DO_AVERAGE, and which is to be freed by END_AVERAGE. Now, the AVERAGE module appears as follows:

```
init_avg: proc returns(ptr);

dcl avg_id ptr;

dcl 1 average_stream based(avg_id),
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

allocate average_stream set(avg_id);
count=0;
total=0;
return(avg_id);
end;  /* init_avg: proc */

average: proc(avg_id,number) returns(fixed bin(15));

dcl avg_id ptr,  /* Points to average data structure. */
    number fixed bin(15);  /* The newest number. */
```

```
dcl 1 average_stream based(avg_id),
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

count=count+1;  /* Another number. */
total=total+number;  /* Total it up. */

return(divide(total,count,15));  /* Return quotient of average. */
end;  /* average: proc */

end_avg: proc(avg_id);

dcl avg_id ptr;

dcl 1 average_stream based(avg_id),
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

free average_stream;
end;  /* end_avg: proc */
```

As with the previous change, because the calling sequence of the subroutine has been changed, the method of calling the subroutine would have to be adjusted for all users of the subroutine, as indicated in the following example:

```
do_average: proc;

dcl current_avg fixed bin(15),
    next_number fixed bin(15),
    avg_id ptr;

dcl tnou entry(char(80),fixed bin(15)),
    tidec entry(fixed bin(15)),
    tovfd$ entry(fixed bin(15)),
    tnoua entry(char(80),fixed bin(15)),
    init_avg entry returns(ptr),
    end_avg entry(ptr),
    average entry(ptr,fixed bin(15)) returns(fixed bin(15));

call tnou('Enter numbers.  Type 0 to stop.',31);
current_avg=0;
next_number=-1;

avg_id=init_avg();

do while(next_number^=0);
   call tnoua('Enter next number: ',19);
   call tidec(next_number);
   if next_number^=0 then current_avg=average(avg_id,next_number);
   end;
```

```
call end_avg(avg_id);

call tnoua('The average is ',15);
call tovfd$(current_avg);
call tnou('',0);

end;
```

This method has an advantage in that, in the future, the AVERAGE module may add information to its AVERAGE_NUMBER based structure without affecting any callers of the module. As with the previous method, full reentrancy is realized even within a single program. To handle two or more simultaneous number streams, the main program needs only make multiple calls to INIT_AVERAGE and END_AVERAGE using different AVG_ID pointers, and use the appropriate pointers in calls to DO_AVERAGE.

Optimizing the General Approach to Conversion: Another example of the ability of the structure hinted at above is that if further optimization is needed when multiple number streams are used, the module can be changed to avoid frequent dynamic allocation of the AVERAGE_STREAM structure. To reduce use of the memory allocation mechanism, a more efficient special-case mechanism can be constructed by reserving storage in the linkage area. This storage, called AVERAGE_STATIC, contains 50 potential copies of AVERAGE_STREAM, where 50 is a number representing a typical figure for the maximum number of simultaneous number streams in use by a particular product.

Of course, reestablishing the use of STATIC data causes the subroutine to become nonreentrant once again. However, additional processing can be performed to manage the single linkage area that the subroutine uses as a process-class subroutine, so that separate program invocations use the same static AVERAGE_STREAM pool, without overwriting each other's data. This produces a form of reentrancy known as active reentrancy. Most PRIMOS system subroutines also practice active reentrancy, using methods similar to the one discussed here.

Causing the AVERAGE module to employ active reentrancy can be done without changing the external calling sequence of the AVERAGE module as most recently shown above. However, major internal changes are needed.

An important consideration is to code the AVERAGE module so that interruption of the module during the management of the STATIC data followed by invocation of another program that called INIT_AVERAGE does not corrupt the linkage area.

When recoded according to these considerations, the AVERAGE module appears as follows:

```
init_avg: proc returns(ptr);

dcl 1 average_static(50) static external,
      2 index_st fixed bin(15) init((50)0),  /* Index within the
                                  array.  -1 means not in array, 0
                                  means not in use, >0 means in use
                                  and is index. */
      2 count_into_based fixed bin(15) init((50)0),
      2 total_into_based fixed bin(31) init((50)0);

dcl 1 average_info static external,
      2 next_index fixed bin(15) init(1),
      2 have_ended_early_indexes bit(1) init('0'b);

dcl 1 average_stream based(avg_id),
      2 index_into_static fixed bin(15),  /* Index within array. */
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

dcl i fixed bin(15),  /* Temporary. */
    avg_id ptr,  /* Pointer to average data structure. */
    cond_store_ok fixed bin(15);  /* 1=Conditional store worked. */

dcl cond_store entry(fixed bin(15),fixed bin(15),fixed bin(15))
           returns(fixed bin(15));

avg_id=null();
do while(avg_id=null());
   i=next_index;  /* See if we can get this element. */
   if i<=hbound(average_static,1)
      then do;  /* Some of array left. */
           next_index=i+1;  /* Either way, increment it. */
           cond_store_ok=cond_store(index_st(i),i,0);
           if cond_store_ok=1
              then avg_id=addr(average_static(i));
           end;  /* Or try again. */
      else do;  /* It isn't easy, consider searching lower. */
           if have_ended_early_indexes  /* Find free element? */
              then do i=1 to hbound(average_static,1);
                      cond_store_ok=cond_store(index_st(i),i,0);
                      if cond_store_ok=1
                         then avg_id=addr(average_static(i));
                      end;  /* Or try again. */
           if avg_id=null()  /* If still not found, allocate. */
              then do;
                   allocate average_stream set(avg_id);
                   index_into_static=-1;  /* Allocated. */
                   end;  /* if avg_id=null() */
           end;  /* if i>hbound(average_static,1) */
   end;  /* do while(avg_id=null()) */
```

```
count=0;
total=0;

return(avg_id);
end;  /* init_avg: proc */

average: proc(avg_id,number) returns(fixed bin(15));

dcl avg_id ptr,  /* Address of the average data structure. */
    number fixed bin(15);  /* The newest number. */

dcl 1 average_stream based(avg_id),
      2 index_into_static fixed bin(15),  /* Index within array. */
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

count=count+1;  /* Another number. */
total=total+number;  /* Total it up. */

return(divide(total,count,15));  /* Return quotient of average. */
end;  /* average: proc */

end_avg: proc(avg_id);

dcl avg_id ptr;

dcl 1 average_static(50) static external,
      2 index_st fixed bin(15) init((50)0),  /* Index within the
                                  array.  -1 means not in array, 0
                                  means not in use, >0 means in use
                                  and is index. */
      2 count_into_based fixed bin(15) init((50)0),
      2 total_into_based fixed bin(31) init((50)0);

dcl 1 average_info static external,
      2 next_index fixed bin(15) init(1),
      2 have_ended_early_indexes bit(1) init('0'b);

dcl 1 average_stream based(avg_id),
      2 index_into_static fixed bin(15),  /* Index within array. */
      2 count fixed bin(15),  /* # of numbers. */
      2 total fixed bin(31);  /* Total value. */

dcl i fixed bin(15),  /* Temporary. */
    cond_store_ok fixed bin(15);  /* 1=conditional store worked. */

dcl cond_store entry(fixed bin(15),fixed bin(15),fixed bin(15))
          returns(fixed bin(15));

if index_into_static>0
   then do;  /* Pointer to within array. */
        i=index_into_static;
        index_st(i)=0;  /* Free again. */
```

```
            cond_store_ok=cond_store(next_index,i,i+1);
            if cond_store_ok=0  /* Wasn't most recent? */
               then have_ended_early_indexes='1'b;
            end;  /* if index_into_static>0 */
         else if index_into_static=-1 then free average_stream;
               else stop;  /* This is an error. */
      end;  /* end_avg: proc */
```

To provide optimal performance, the INIT_AVERAGE and END_AVERAGE procedures have undergone extensive changes. However, the AVERAGE procedure itself has remained unchanged, except for the new declaration of the average data structure. AVERAGE does not need to consider whether the pointer passed to it identifies storage within the linkage area or dynamically obtained storage.

Instead, INIT_AVERAGE and END_AVERAGE together manage the external static (common) storage that is used for quick "allocation" of AVERAGE_STREAM structures. The AVERAGE_STATIC array contains 50 copies of potential AVERAGE_STREAM structures. Now included in each structure is an INDEX that identifies whether the structure is available or not (within the array), or separately allocated (not within the array). Separate allocation occurs only when the array of AVERAGE_STATIC is fully used, and hence represents a graceful degradation when more than 50 number streams are in use at a time.

An important subroutine used in the newest AVERAGE module is the COND_STORE subroutine. This subroutine is a PMA module that takes a FIXED BIN(15) location in memory, a new value for that location, and an old value for that location. The COND_STORE subroutine updates the FIXED BIN(15) location to the specified new value only if the old value is accurate. Otherwise, it leaves the location unchanged. It returns a 1 if it succeeds in updating the location; 0 if it does not.

Because it uses the STAC machine instruction, the verification of the old value and update of the new value are guaranteed to occur in an atomic fashion, independent of any other processing on the system.

A listing of COND_STORE.PMA is found in Chapter 8, as it is a useful subroutine for updating information in shared memory.

Methods such as those listed above may seem extensive, but they can result in better throughput for your product. In addition, when the target of such a procedure includes a number of separate subroutines that manage a single common area, folding them into one procedure with alternate entrypoints may improve the maintainability of that portion of your product.

Determining the Class Requirements of Your Library EPF

Once you have determined the requirements for all of the subroutines in the library EPF you wish to build, decide whether you will have one library EPF, either process-class or program-class, or two library EPFs, one of each class. On a per-subroutine basis, the decisions are:

- A subroutine that must be in one particular class must be placed in a library EPF of that particular class.
- A subroutine that can operate in either class can be placed in either class, but will probably operate more efficiently when placed in a process-class library EPF.

If you have subroutines in each class that require being in that particular class, then you must have two library EPFs, one of each class.

If all your subroutines require being in the program class, or they all require being in the process class, then you must have one library EPF of the appropriate class. It is rare for a subroutine to require being in the process class, although dependencies on being in the program class are common.

If none of your subroutines require being in the program class, then you need create only one process-class library EPF. This is the most desirable situation, as it results in the best performance.

Typically, however, you will have some subroutines that require being in the program class, and others that can be in either class. In this case, you can create either one program-class library EPF, or two library EPFs (one of each type).

The tradeoff depends on the overhead of having PRIMOS maintain information on a second library EPF, including a search list entry, certain internal resources on a per-library EPF basis, and two separate linkage initialization phases for using your library as a whole.

If this overhead is less than the overhead of having PRIMOS reallocate and reinitialize linkage areas belonging to subroutines that do not require the program class, then having two library EPFs represents a reasonable performance tradeoff.

Otherwise, the added overhead of a second library EPF for your library is not worth the savings of separating the linkage areas, so one program-class library EPF should be used.

Note

Because PRIMOS does not support dynamic linking to common areas, you must place all subroutines that reference a particular common area in the same EPF as the common area itself. For example, if subroutines A and B wish to

communicate via a common area named IOBUF, then both subroutines A and B and common area IOBUF must all be linked into one EPF. If instead you place subroutine A in a program-class library EPF and subroutine B in a process-class library EPF, for example, then they each get their own copies of IOBUF and therefore cannot communicate with each other through IOBUF.

HOW TO USE DBG ON A LIBRARY EPF

Debugging a library EPF using DBG requires that the object (.BIN) files that comprise the library EPF be linked into a program EPF along with a subroutine that serves as as the main entrypoint for the program EPF. Typically, the main subroutine entrypoint is used only for debugging and testing of the library EPF, and it does not necessarily require any code. It should, however, contain declarations for all subroutine entrypoints in the library EPF.

You may find it useful to also declare storage in the main entrypoint of the program EPF, to be used during DBG CALL commands to subroutine entrypoints as storage for the input and output arguments. If you do this, you should have the main entrypoint initialize all its variables so that an attempt by the user of DBG to examine some of the data (such as varying character strings) does not produce garbage output on the screen.

The last statement executed by such a test bed is, in PL1/G:

```
CALL SIGNL$('PAUSE$',NULL(),0,NULL(),0,'C000'b4);
```

In FTN or F77, use a PAUSE statement.

This signals a condition that is intercepted by DBG, causing DBG to enter subcommand level without finishing execution of the program. (When the main program finishes, access to the variables declared within it are lost.)

A sample main entrypoint subroutine might be:

```
test_bed_for_xyz_library: proc;

/* Declare all the library entrypoints. */

dcl xyz_input_coordinates entry(fixed bin(15),fixed bin(15)),
    xyz_plot entry(ptr,fixed bin(15),fixed bin(15)),
    xyz_update entry(ptr),
    xyz_output_graph entry(ptr),
    xyz_delete_user entry(char(32) var);
```

```
/* Declare storage used for library entrypoints. */

dcl x fixed bin(15),  /* For XYZ_INPUT_COORDINATES. */
    y fixed bin(15),
    graph_ptr ptr,  /* For XYZ_UPDATE and XYZ_OUTPUT_GRAPH. */
    user_name char(32) var;  /* For XYZ_DELETE_USER. */

/* Declare graph structure. */

dcl graph(24,80) char(1);

/* Declare SIGNL$ subroutine. */

dcl signl$ entry(char(32) var,ptr,fixed bin(15),ptr,fixed bin(15),
          bit(16));

/* Initialize variables. */

x=0;
y=0;
graph_ptr=addr(graph);
user_name='';
graph=' ';

/* Now pause, invoking DBG subcommand level. */

call signl$('PAUSE$',null(),0,null(),0,'C000'b4);

/* If user continues, run the game. */

x=1;
do while(x^=0);
   call xyz_input_coordinates(x,y);
   if x>=1 & x<=80 & y>=1 & y<=24
      then call xyz_plot(graph_ptr,x,y);
   end;  /* do while(x^=0) */

call xyz_output_graph(graph_ptr);
do while('1'b);
   call signl$('PAUSE$',null(),0,null(),0,'C000'b4);
   call xyz_update(graph_ptr);
   call xyz_output_graph(graph_ptr);
   end;

/* End of test bed. */

end;
```

Build the test bed program EPF by linking the main test bed subroutine with the object files used that comprise the library EPF. Then, type:

```
DBG test-bed-program
RESTART
```

The RESTART command causes the work area variables to be initialized. The main program is then suspended, returning you to DBG mode. At this point, you can either CONTINUE the main program to perform typical system or unit tests, or use the CALL subcommand to test the behavior of specific subroutines in your library.

ENTRYPOINT SEARCH LISTS

PRIMOS makes the connection between a library EPF and a program that wishes to use the library EPF when the program attempts to call a subroutine in that library EPF. This causes the dynamic linking mechanism to be invoked. The dynamic linking mechanism is described in Chapter 2.

When the dynamic linking mechanism is invoked, it first searches the list of internal PRIMOS entrypoints. If the desired subroutine is not found there, the dynamic linking mechanism uses an entrypoint search list to direct it to library EPFs that are to be searched for the desired subroutine. In addition, the search list specifies at what point the static-mode libraries are to be searched, if they are to be searched.

Each user has an in-memory copy of an entrypoint search list. This in-memory copy is loaded from a file on disk with the name ENTRY$.SR or with a name ending in .ENTRY$.SR. The file is loaded into memory either:

- When the first dynamic link for a user is encountered
- When a user issues the SET_SEARCH_RULES command (abbreviated SSR)

If the first dynamic link is encountered before a user issues the SET_SEARCH_RULES command after logging in, PRIMOS loads the default entrypoint search list, which has the pathname SYSTEM>ENTRY$.SR.

A user may use the SET_SEARCH_RULES command to switch to a new entrypoint search list or to return to the default entrypoint search list.

Typically, the default entrypoint search list indicates that system-wide library EPFs (in the LIBRARIES* UFD) are to be searched first (after internal PRIMOS entrypoints, which are always searched before any libraries listed in the entrypoint search list). These

libraries include the system library (SYSTEM_LIBRARY), the FORTRAN I/O library (FORTRAN_IO_LIBRARY), the application library (APPLICATION_LIBRARY), and so on.

At some point, the default search list usually directs that the static-mode libraries are to be searched. Although Prime supplies several individual static-mode libraries, these libraries are treated by the search list mechanism as one library. If the desired subroutine is still not found, the default search list may specify further library EPFs that are to be searched. If the end of the search list is reached and the target subroutine has still not been found, the dynamic linking mechanism signals the condition LINKAGE_FAULT$, which typically produces an error message such as:

```
Error: condition "LINKAGE_FAULT$" raised at 4243(3)/1031.
Entry name "INIT_LINE" not found while attempting to resolve
dynamic link from procedure "TRY_ASYNC" .
ER!
```

An entrypoint search list consists of one or more search rules. A particular line within a search list is referred to as a search rule (singular).

To display your current search list, use the LIST_SEARCH_RULES command (abbreviated LSR). For example:

```
OK, LIST_SEARCH_RULES

Pathname of template: <SYSDSK>SYSTEM>ENTRY$.SR

    LIBRARIES*>SYSTEM_LIBRARY.RUN
    LIBRARIES*>APPLICATION_LIBRARY.RUN
    -STATIC_MODE_LIBRARIES
    LIBRARIES*>PL1G_LIBRARY.RUN
    LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
    LIBRARIES*>COMMON_ENVELOPE.RUN
    LIBRARIES*>OPTIMIZER.RUN
    LIBRARIES*>CODEGEN_COMMON.RUN
    LIBRARIES*>CODEGENV.RUN
    LIBRARIES*>CODEGENI.RUN
    LIBRARIES*>CBL_LIBRARY.RUN
    LIBRARIES*>CC_LIBRARY.RUN
    LIBRARIES*>PASCAL_LIBRARY.RUN
    LIBRARIES*>VRPG_LIBRARY.RUN
OK,
```

As shown in the above sample, the default system search list comes from the file SYSTEM>ENTRY$.SR. All entrypoint search rule files must be named ENTRY$.SR, or end in .ENTRY$.SR, to identify them as library entrypoint search rule files.

Notice that the static-mode libraries are identified using the -STATIC_MODE_LIBRARIES option.

The file containing the search rules, SYSTEM>ENTRY$.SR, is a text file that contains the lines shown in the above sample. As with other text files, the file can be modified by using text editors such as ED or EMACS. The file corresponding to the above sample search list is shown below:

```
LIBRARIES*>SYSTEM_LIBRARY.RUN
LIBRARIES*>APPLICATION_LIBRARY.RUN
-STATIC_MODE_LIBRARIES
LIBRARIES*>PL1G_LIBRARY.RUN
LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
LIBRARIES*>COMMON_ENVELOPE.RUN
LIBRARIES*>OPTIMIZER.RUN
LIBRARIES*>CODEGEN_COMMON.RUN
LIBRARIES*>CODEGENV.RUN
LIBRARIES*>CODEGENI.RUN
LIBRARIES*>CBL_LIBRARY.RUN
LIBRARIES*>CC_LIBRARY.RUN
LIBRARIES*>PASCAL_LIBRARY.RUN
LIBRARIES*>VRPG_LIBRARY.RUN
```

The ordering of individual rules is important, as it reflects the order in which the libraries are searched. The search order is important for performance reasons, as frequently-called subroutines (such as the subroutines in SYSTEM_LIBRARY) should require the shortest search time possible. In addition, the search order is important when naming conflicts occur between libraries — the order in which the conflicting libraries appear decides which copy of a subroutine is actually invoked.

Setting Your Own Search Rules

Setting your own private search rules is important when you are developing a library EPF. Only by updating your search rules do you enable programs that you run to be able to call, or link to, subroutines in your library EPF. This section discusses how to set your own search rules.

First, you create your own search rule file. It must be named ENTRY$.SR, or end in .ENTRY$.SR, to be considered a valid entrypoint search rules file.

The simplest form of this file is:

```
-SYSTEM
MYUFD>MYLIBRARY.RUN
```

The -SYSTEM rule specifies that the standard system search rules, found in SYSTEM>ENTRY$.SR, are to be searched, and then the library EPF specified is to be searched. The contents of SYSTEM>ENTRY$.SR replace the -SYSTEM rule in memory at the time you issue the SET_SEARCH_RULES command, not later when dynamic linking takes place.

Keep in mind that these lists are searched only when the dynamic linking mechanism is actually invoked. Until a subroutine in your library EPF is called, it may not necessarily be mapped into memory.

Note

The -SYSTEM rule affects the ordering of the search only if the -NO_SYSTEM option is specified on the SET_SEARCH_RULES command line, as described below. Otherwise, -SYSTEM is ignored, and the search rules in the default entrypoint search list are used before the search rules in the personal entrypoint search list.

Although -SYSTEM is ignored, you should place it in the file as shown, just in case somebody accidentally specifies the -NO_SYSTEM option on the SET_SEARCH_RULES command line. The side effects of having only one search rule in an entrypoint search list are very strange and difficult to identify.

A more complex method of creating your search rules file is to actually list all of the libraries to be searched, in the order you desire. For example:

```
MYUFD>MYLIBRARY.RUN
LIBRARIES*>SYSTEM_LIBRARY.RUN
LIBRARIES*>FORTRAN_LIBRARY.RUN
LIBRARIES*>FORTRAN_IO_LIBRARY.RUN
LIBRARIES*>APPLICATION_LIBRARY.RUN
-STATIC_MODE_LIBRARIES
```

This has an effect similar to the previous search rules file, except that the library EPF MYUFD>MYLIBRARY.RUN is searched before any other library is searched, with the exception of internal PRIMOS entrypoints.

Once the search rules file is created, you establish it for the duration of your login session using the SET_SEARCH_RULES command.

The format of this command is:

```
{ SET_SEARCH_RULES } search-rules-file.ENTRY$ [-NO_SYSTEM]
{ SSR             }
```

This changes your current search list to the search rules specified in search-rules-file.ENTRY$.SR. You will revert to the standard system search rules (in SYSTEM>ENTRY$.SR) only when you log in again, when your command environment is reinitialized, or when you issue the command:

```
SET_SEARCH_RULES -DEFAULT ENTRY$
```

To make changes to your search list permanent, update your LOGIN.CPL or LOGIN.COMI file to include the appropriate SET_SEARCH_RULES command.

The -NO_SYSTEM option, when present, indicates that the default system search list, SYSTEM>ENTRY$.SR, is not to be automatically inserted in front of your search rule when you issue the SET_SEARCH_RULES command. You should use this option only if you have used a copy of the default system search list to build your own search rule. Careless use of the -NO_SYSTEM option may cause erratic behavior in Prime-supplied programs.

> Caution
>
> Do not use the SET_SEARCH_RULES command while there are active program or library EPFs in your command environment, except to add new library EPFs that you are developing to the end of the list (after the -SYSTEM rule). It is recommended that you use the INITIALIZE_COMMAND_ENVIRONMENT command before each use of SET_SEARCH_RULES. It is not intended for the active search list of a user to be frequently changed. Deleting or modifying particular rules in the search list while active EPFs abound is likely to cause inconsistent program behavior.

If your login program (LOGIN.CPL, LOGIN.COMI, LOGIN.RUN, or LOGIN.SAVE) sets up your search rules, then use of the INITIALIZE_COMMAND_ENVIRONMENT command, which invokes your login program, also causes your search rules to be set up.

In this case, either modify your personal copy of the search rules so that your login program uses the correct copy, or modify your login program to use the new copy that you have constructed.

Advanced Use of Entrypoint Search Lists

In some cases, there may be three levels of entrypoint search list activity:

- The system-wide search list, maintained by the System Administrator
- The search list for a particular application or project, which changes relatively frequently
- A user who has his or her own library EPFs and who therefore has a personal search list, yet needs to also utilize the other two search lists

In this case, the user might have difficulty maintaining his or her personal search list so that it reflected the latest changes for the application or project he or she is associated with.

A solution to the problem is to place search rules specific to the application or project in a central entrypoint search list file. Then, personal search lists can use the special rule -USE to refer to the project list.

For example, a project-wide entrypoint search list file named PROJECT_A>ENTRY$.SR might read:

```
PROJECT_A>ASYNC_LINE.RUN
PROJECT_A>X.25_COMMS.RUN
PROJECT_A>SCREEN_FORMS.RUN
```

A personal entrypoint search list file might then read:

```
-SYSTEM
-USE PROJECT_A>ENTRY$
MYDIR>MYLIBRARY.RUN
```

Anytime the SET_SEARCH_RULES command is issued for the personal entrypoint search list file, the project-wide entrypoint search list file is automatically included.

This allows the project leader of PROJECT_A to change the project-wide search list file without having to ask users on the system who have their own personal entrypoint search list files to update their files.

EXAMINING ENTRYPOINT LISTS

You can examine the list of entrypoints for either a library EPF or a particular library (.BIN) file.

Examining Entrypoints in a Library EPF

You may examine the list of entrypoints for a library EPF by using the LIST_LIBRARY_ENTRIES command (abbreviated LLENT). This is useful when:

- You want to check other library EPFs use at your installation for possible conflicts with names you intend to use as entrypoint names
- You want to check that you have declared all entrypoints in your own library EPF correctly
- You want to determine in which library EPF a particular entrypoint exists

Checking a Particular Library EPF

To check a particular library EPF for whether it declares a particular name as an entrypoint, or to list all of its entrypoints, use the command:

```
LIST_LIBRARY_ENTRIES epf-filename [-ENTRYNAME name ...]
```

It does not matter whether the library EPF you are checking is yours or not.

The library EPF is epf-filename. You may specify as many as eight entrypoint names (name) to search for, or you may leave off the -ENTRYNAME specification to display all of the entrypoints for the library.

Locating a Particular Entrypoint

To determine in which library EPF a particular entrypoint exists, use the following form of the LIST_LIBRARY_ENTRIES command:

```
LIST_LIBRARY_ENTRIES -ENTRYNAME name ...
```

All library EPFs specified in your entrypoint search list are checked for the existence of the entrypoint named name. (You may specify as many as eight entrypoint names.)

If any library EPFs specified in your entrypoint search list are inaccessible for some reason, such as when a library EPF does not not exist in the specified directory, LIST_LIBRARY_ENTRIES displays an error message. Therefore, LIST_LIBRARY_ENTRIES may be used to verify the correctness of an entrypoint search list.

THE LIBRARY EPF MECHANISM

This section describes specific aspects of the EPF mechanism that apply only to library EPFs. Thorough familiarity with the description of the EPF mechanism found in Chapter 3 is a prerequisite for understanding this section.

Specifically, this section describes:

- The automatic mapping of library EPFs during dynamic linking
- How PRIMOS decides whether to skip Phases 5 and 6 of the EPF mechanism for a library EPF
- Storage allocation issues relating to library EPFs

The Automatic Mapping of Library EPFs

The only way a library EPF is accessed by another program is by encountering a dynamic link (via a faulted IP) that identifies, as its target, an entrypoint in the library EPF.

The dynamic linking mechanism, described in Chapter 2, detects the faulted IP. After it has searched the internal PRIMOS entrypoints for the desired subroutine, it uses the user's entrypoint search list to determine where to look for the desired subroutine next.

Each rule in the entrypoint search list is either:

- The pathname of a library EPF
- The rule -STATIC_MODE_LIBRARIES which indicates that the static-mode libraries are to be searched
- The rule -SYSTEM which indicates that the default entrypoint search list, SYSTEM>ENTRY$.SR is to be searched; this list has in it only rules of the first two types listed above

When the dynamic linking mechanism encounters a pathname rule, it checks to see whether the library EPF identified by the pathname is already mapped in. If it is not, the dynamic linking mechanism must map the library EPF into memory (Phase 4 of the life of an EPF, as presented in Chapter 3) before it can search its list of entrypoints.

Once the library EPF is mapped in, the dynamic linking mechanism searches its list of entrypoint names to see if it contains the desired subroutine. If not, the library EPF remains mapped-in but inactive. A library EPF is removed only by a user command (such as REMOVE_EPF) or a call to the EPF$DEL subroutine.

If the library EPF does contain the desired subroutine, the dynamic linking mechanism does not need to map it in because it has already been mapped in. Instead, the dynamic linking mechanism checks to see if it needs to allocate and initialize linkage for the EPF, as described next.

Phases 5 and 6 of the EPF Mechanism

In the life of an EPF, presented in Chapter 3, Phases 5 and 6 are:

5. The linkage (impure) portion of the EPF is allocated (EPF$ALLC).

6. The linkage (impure) portion of the EPF is initialized (EPF$INIT).

Each time the dynamic linking mechanism processes a dynamic link to a library EPF, it must determine whether it needs to allocate and initialize the linkage (impure) portions of that EPF. This decision is primarily based upon whether the library EPF is program-class or process-class, and whether it is already in use (by the program or by the process).

Program-Class Library EPF: For a program-class library EPF, the dynamic linking mechanism checks to see whether the program invoking the library EPF has already linked to the same EPF. If it has, then the impure portions of the library EPF that correspond to the program have already been allocated and initialized, and Phases 5 and 6 are skipped.

The impure portions of a process-class library EPF are deallocated whenever the user changes command levels, unless the library EPF is in use by a suspended program. (This behaviour may change at future Revisions of PRIMOS.)

Process-Class Library EPF: For a process-class library EPF, the dynamic linking mechanism checks to see whether the EPF has already had its impure portions allocated and initialized. If it has, then Phases 5 and 6 are skipped. Even if a different program caused the allocation and initialization of the EPF, Phases 5 and 6 are skipped, because only one copy of the impure portions of a process-class library EPF are kept for a user, and they are not reinitialized when a new program starts using the EPF.

Impure portions of a process-class library EPF are deallocated when:

- The user logs out.
- The user explicitly removes the process-class library EPF by using the REMOVE_EPF command.
- The user's command environment is reinitialized, either explicitly (via the INITIALIZE_COMMAND_ENVIRONMENT command) or implicitly (as a result of an error condition detected by the command environment and identified as being unresolvable).

However, a process-class library EPF that is still in use by a suspended or running program cannot have its impure portions deallocated.

Storage Allocation Issues

The Prime 50-Series architecture allows the dynamic allocation of stack space during procedure call. In addition, PRIMOS allows the dynamic allocation and deallocation of memory via explicit requests by a running program.

Dynamic memory is allocated during program runtime as a result of either:

- Compiler-generated requests for temporary storage, such as for the storing of a temporary character string during the execution of a string concatenation operation
- Program-directed requests for memory, such as via the ALLOCATE statement in PL1 Subset G

Normally, memory dynamically allocated by a program is automatically deallocated (freed) by PRIMOS when the program terminates. In addition, any memory dynamically allocated by program-class library EPFs invoked by that program is also deallocated.

However, memory dynamically allocated by a process-class library EPF must not be deallocated by PRIMOS when a program terminates. This is because the linkage portion of that EPF, which may contain pointers to the dynamically allocated memory, is not deallocated; it is instead reused.

Therefore, PRIMOS must distinguish between a program-class library EPF and a process-class library EPF when allocating memory to ensure that it does not, later, automatically deallocate memory acquired by a process-class library EPF.

This distinction is made during the linking of the library EPF. For a process-class library EPF, using the LIBRARY PROCESS_CLASS subcommand specifies that dynamically allocated memory is to be acquired from a special memory pool, called process-class storage. No memory from this pool is ever explicitly deallocated by PRIMOS except during logout and command environment initialization.

If the LIBRARY PROCESS_CLASS subcommand is not used, as for a program-class library EPF, dynamically allocated memory is acquired from the program-class storage pool used by program EPFs. Memory allocated from this pool by a particular program is automatically deallocated by PRIMOS when the program terminates.

If a process-class library EPF is built without the LIBRARY PROCESS_CLASS subcommand, then any language-driven allocation, either explicitly via statements such as ALLOCATE in PL1/G, or implicitly via compiler-generated allocation for temporary storage, will fail when the library EPF executes. The failure will be in the form of a LINKAGE_ERROR$ condition raised. The condition is raised because the process-class library EPF attempted to link to a program-class library EPF in which the program-class allocator resides.

Caution

A pointer to storage that has been dynamically allocated as program-based storage should not be passed to a process-class subroutine if that subroutine stores the pointer in linkage area or in dynamically allocated memory. Similarly, the address of a program-class entrypoint should not be passed to a process-class subroutine unless the subroutine stops using the address when it returns to its caller.

In general, a pointer to object A should never be passed to subroutine B if the life-span of the storage used by subroutine B to hold the pointer to object A may exceed the life-span of object A itself. Otherwise, the termination of object A followed by the continued execution of subroutine B may result in the reference by B to the (nonexistant) object A, producing unpredictable (and invariably incorrect) results.

While this is a general principle of programming methodologies, it applies specifically to the interactions between program-class subroutines and process-class subroutines.

7

Coding Guidelines for EPFs

This chapter describes the coding guidelines that you should follow when writing subroutines or main programs that are going to be built as EPFs. None of these guidelines preclude the use of these subroutines and main programs in static-mode applications built with SEG except as otherwise noted.

Specifically, this chapter describes:

- How to write modules in Prime-supplied high-level languages for EPFs
- How to write modules in PMA (Prime Macro Assembler) for EPFs

WRITING MODULES IN HIGH-LEVEL LANGUAGES FOR EPFS

Most Prime-supplied language compilers produce EPF-compatible object (.BIN) files. These languages include:

- F77
- FTN (when using the -64V or -DYNM options)
- Pascal
- PL1/G
- VRPG

- CBL
- C

Using these compilers always produces EPF-compatible object files. An exception is the FTN compiler when the -PBECB option is specified. If -PBECB is specified on the FTN command line, BIND produces a warning message when the generated object file is linked:

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

If this warning message appears, it indicates that the EPF is not likely to successfully execute when invoked.

When the -PBECB option is used with compilers other than FTN (for those compilers that support the option), the compilers mark the compiled modules as impure. BIND places procedure text for impure modules in the linkage area in specially marked segments called IMPURE segments. This allows PRIMOS to modify the ECBs when the program is executed, while preventing the procedure text for such modules from being shared between users and from being mapped directly from the file system disk. Therefore, while -PBECB may enhance the performance of a shared static-mode application, it typically reduces the performance of an EPF.

Writing the Main Entrypoint of a Program EPF

You write the main entrypoint of a program EPF exactly as you would write a subroutine. You may use the PROGRAM statement (instead of the SUBROUTINE statement) in F77 or the OPTIONS(MAIN) keyword (on the PROCEDURE statement) in PL1/G if you desire. However, neither of these conventions is required for a main entrypoint. Requirements for the calling sequence of main entrypoints are described in the Programmer's Guide to BIND and EPFs and in Volume III of this series.

Note that a program built via SEG in a fashion that produces a RESUMEable static-mode runfile typically requires the main entrypoint to be named MAIN. No such requirement exists for BIND and EPFs; however, you may wish to keep the SEG requirement in mind if you intend to use a main entrypoint in both the EPF and static-mode environments.

WRITING MODULES IN PMA FOR EPFS

This section summarizes basic concepts of PMA (Prime Macro Assembler) programming, and then discusses specific requirements for writing PMA subroutines that are to execute as EPFs.

Basic Concepts of PMA Programming

A PMA source file is referred to as a module. It may contain one or more subroutines. When a module is assembled (using PMA), an object file is generated, usually with the .BIN suffix on its file name. This object text consists of:

- Module description information
- Procedure text for each subroutine
- Linkage text for each subroutine
- Stack and parameter allocation information for each subroutine entrypoint
- Linkage information for each subroutine entrypoint
- External linkage information, including references to common areas and other subroutines

You tell PMA which part of the module you are building by including special pseudo instructions in the PMA source code. Pseudo instructions are directives to the assembler; usually, they change the way in which subsequent lines in the source file are interpreted. Pseudo instructions themselves may or may not cause specific data (such as instructions or storage allocation information) to be generated in the object text.

All PMA modules must have the END pseudo instruction as the last line in the file. PMA modules that serve as main entrypoints for a program (whether an EPF or a static-mode program) must name the main entrypoint's ECB in the operand field of the END pseudo instruction. No comment lines or blank lines may follow the END pseudo instruction. Other important pseudo instructions are described below.

PMA subroutines that are to be linked into EPFs are usually constructed according to the following template:

Source Text	Meaning
*	Comment lines describing the subroutine
SEG or SEGR	Pseudo instruction to specify a V-mode or I-mode module
SYML	Optional pseudo instruction to turn on long (as many as 32 character) symbol names
RLIT	Optional pseudo instruction to cause placement of literals in procedure text

ENT	Optional pseudo instructions to export names for reference by external modules
LINK	Pseudo instruction to switch to linkage text generation for placement of the ECB
ECB	Pseudo instruction to generate the ECB itself, and, optionally, additional ECBs for alternate entrypoints or internal subroutines
DYNM	Pseudo instructions to specify stack frame allocation
EXT	Optional pseudo instructions to specify external symbols
PROC	Pseudo instruction to switch back to generating procedure text
instructions	The procedure code of the module
LINK	Optional pseudo instruction for switching to linkage text generation
data	Various address definition, data definition, and storage allocation pseudo instructions (optional), to describe the format and data for the link frame
END	Pseudo instruction to delimit the end of the module and optionally designate the main entrypoint of the module

The remainder of this section describes portions of the object text and of the above template that are specifically related to coding a PMA module for execution within an EPF. See the Assembly Language Programmer's Guide for further information on PMA. For information on the instruction sets and architecture of the Prime 50 Series machines, see the System Architecture Reference Guide.

Use of SEG or SEGR: The first non-comment line of a proper PMA subroutine must be either the SEG pseudo instruction (for a V-mode subroutine) or the SEGR pseudo instruction (for an I-mode subroutine). If this is not the case, BIND refuses to link the object text (.BIN file) generated by assembling the subroutine via PMA. Additionally, the keyword PURE or IMPURE should follow the SEG or SEGR keyword on the same line, as described below in the sections on Impure PMA Module Restrictions and Pure PMA Modules. If neither PURE nor IMPURE is specified, the default is PURE.

Procedure Text: The procedure text for a subroutine consists of the instructions that make up the body of the subroutine. In a PMA subroutine, procedure text generation is specified via the pseudo instruction:

```
PROC
```

Linkage Text: The linkage text for a subroutine consists of static data used and modified by the subroutine. Only one copy of linkage text exists for a subroutine within a program or library, even if the subroutine invokes itself recursively. Linkage text generation is specified via the pseudo instruction:

```
LINK
```

Stack and Parameter Allocation Information: The DYNM pseudo instruction is used to specify the allocation of the stack frame for the module. The stack frame is also used to hold the argument list pointers for the subroutine invocation. Each subroutine invocation causes the dynamic allocation of its stack frame. Initially, a stack frame contains undefined values except for the stack frame header and the argument pointers (if any).

Typically, the DYNM pseudo instruction is used in the following manner:

```
DYNM temporary-1(size-1),temporary-2(size-2)
DYNM argument-1(3),argument-2(3),argument-3(3),argument-4(3)
DYNM temporary-3(size-3),temporary-4(size-4)
```

The argument list pointers must be allocated 3 halfwords each, and must be contiguous in the stack frame as indicated. Other temporaries can precede or follow the argument list template in the stack frame. The start ot the argument list template in this case is argument-1, and the number of arguments is 4.

The DYNM pseudo instruction provides the only way of allocating stack frame storage in PMA. Using an EQU pseudo instruction to set a symbol equivalent to, say, SB%+102 does not affect allocation of the stack frame in any way.

Use of DYNM changes allocation of storage to the stack frame only temporarily. The current assembly pointer is still either in procedure or linkage text, so machine instructions and data generation directives following DYNM are placed in either the procedure or the linkage area rather than the stack frame. (You cannot specify initial values for storage in the stack frame except by including prologue code in your subroutine to perform the initialization at runtime.)

Linkage Information: Information for each subroutine entrypoint that describes the entrypoint to BIND is called linkage information. Its purpose is to tie together the procedure text, linkage text, and stack and parameter allocation information for the entrypoint.

This information is turned into an ECB (Entry Control Block) for the entrypoint by BIND. When the EPF is invoked, PRIMOS modifies the ECB for each subroutine in the EPF so that the pointer to the linkage text in each ECB identifies the actual location of the linkage text. For this reason, a PMA subroutine that has its ECB in the procedure text behaves as an impure subroutine if it is linked into an EPF via BIND.

Linkage information is declared via the ECB pseudo instruction. Here is a sample use of the ECB pseudo instruction:

```
            LINK
ecb_label ECB  first_instruction_label,,first_arg,n_args
```

This ECB is placed in the linkage text for the module. The label for it is ecb_label and identifies the actual target of procedure call (PCL) instructions to the entrypoint. The ENT pseudo instruction is used to associate the exported (externally available) symbol name with ecb_label:

```
ENT external_name,ecb_label
```

If external_name and ecb_label are the same name, then only ENT ecb_label need be specified.

The label of the first instruction to be executed (an ARGT instruction when the procedure has one or more arguments) is identified via first_instruction_label. The label of the start of the argument list template is first_arg and must refer to a stack-relative label (declared via the DYNM pseudo instruction). The number of arguments is specified as n_args.

The ECB pseudo instruction can be used to specify other information not described above. For example, between the two commas in the form above, you could define the start of linkage text for the entrypoint. It defaults to the start of linkage text for the module, as indicated via the LINK pseudo instruction. Another optional field, the stack size, defaults to the amount of stack space explicitly reserved via the DYNM pseudo instruction. You can also specify the initial value of the keys register via ECB, although it defaults (appropriately) to the addressing mode of the module (V-mode or I-mode).

External Linkage Information: A PMA module must often refer to symbols that are not defined within the scope of the module itself. These are called external references.

A reference to a subroutine that is defined externally is a reference to an external subroutine. For the most part, references to external subroutines are handled automatically by PMA via the CALL pseudo instruction:

```
CALL subroutine
```

When PMA detects a CALL pseudo operation while assembling V-mode or I-mode code, it:

- Identifies subroutine as an external reference, as if the pseudo instruction EXT subroutine had been issued
- Places one IP (Indirect Pointer) in the linkage text that points to the external subroutine at runtime, for use by all CALLs to that subroutine in the current module, as if the following instruction sequence had been present:

```
                LINK  (Switches to generating linkage text)
subroutine_ip IP subroutine
                PROC  (Only if originally in procedure text)
```

- Generates a procedure call instruction to invoke the subroutine, identifying indirection through the IP it generated as the target of the instruction:

```
PCL subroutine_ip,*
```

All of the above can be explicitly specified by the PMA programmer, but use of the CALL pseudo instruction is recommended when calling external subroutines. (To call a subroutine within the current module, use a PCL to its ECB without specifying indirection.)

Another form of external reference includes references to program common areas and other symbols. Here, PMA also automatically generates IPs and implicitly forms indirect instructions that refer to the external symbols. However, the symbols must be explicitly declared as external as follows:

```
EXT symbol
```

> Caution
>
> Do not use the XAC pseudo instruction or its equivalent EXT/DAC pair in V-mode or I-mode PMA modules. PMA does not treat this usage as an error; however, neither BIND nor SEG support that form of external link (DAC and XAC generate only a 16-bit halfword link), and PRIMOS does not support the conversion of an imaginary 16-bit address to an actual 16-bit address.

The COMM pseudo instruction is particularly useful for building representations of common areas. PMA automatically generates IPs for references into common areas, including references into the midst of common areas. In other words, PMA does not generate a single IP to the beginning of a common area and then use offset addressing (via the XB or X registers) to access items within the common area. Instead, PMA generates one IP for each reference into a common area at a different offset. This method produces more efficient code in terms of execution time at the expense of the size of linkage text (as more than one IP may be needed to access each common area). It also allows PMA to avoid making de facto use of the XB or X register, either or both of which may be used by the programmer in neighboring instructions. However, because PMA must convert instructions referencing common areas so that they go indirect through IPs, the instructions in the source program cannot specify indirection.

If you want to refer to items within a common area using offset addressing rather than directly through an IP, you must use either the XB or X register. To use the XB register, code the instruction:

```
EAXB common_area    (Becomes common_area_ip,*)
```

Then, your program performs subsequent references to items within the common area by referencing XB%+offset, where offset is the offset of the item, in halfwords, from the beginning of common_area.

To use the X register, code the instruction:

```
LDX =offset
```

Subsequent references to the item that is offset halfwords from the beginning of common_area are performed by referencing common_area,X. For example:

```
LDA common_area,X
```

Because common_area is an external, PMA automatically translates this into:

```
LDA common_area_ip,*X
```

Therefore, you cannot perform indirection through a pointer in a common area without using effective address calculation and the XB register.

Note

When using the XB or X register, remember that, as with all other general-purpose registers, the PCL (also CALL) instruction may destroy the register contents.

Designating the Main Entrypoint: If you are writing a PMA module that is to contain the main entrypoint for a program EPF, you must designate the main entrypoint of the module by specifying the symbol name for the ECB in the operand field of the END pseudo instruction at the end of the module. For example:

```
          SEG
          RLIT
          SYML
*
          SUBR  COUNT,COUNT_ECB
*
          LINK
COUNT_ECB ECB   COUNT_START,,COMMAND_LINE,2
*
          DYNM  COMMAND_LINE(3),SEVERITY_CODE(3)
*
          PROC
*
COUNT_START EQU *
          ARGT

          .
          .
          .

          END   COUNT_ECB
```

As this example illustrates, you must specify the label that tags the ECB for the main entrypoint (COUNT_ECB), not the external name of the subroutine (COUNT) or the starting address of the procedure code (COUNT_START).

If you specify the main entrypoint in this fashion, you may still use the module as a subroutine rather than a main program; in this case, your specification of the main entrypoint is ignored.

If you fail to specify the main entrypoint as shown, linking the assembled module as the first module in a program EPF produces an EPF that, when run, might produce an error message such as:

```
Error: condition "ILLEGAL_SEGNO$" raised at 41(3)/122722.
(Referencing 1(3)/0).
ER!
```

If you do not have access to the source code of the module, or if you wish to use a "quick fix", relink the module and use the MAIN subcommand of BIND to specify the entrypoint of the module that is the main entrypoint of the program EPF. You may do this particularly quickly by using the following command sequence:

```
BIND
LOAD failing-program.RUN
MAIN main-entrypoint-name
FILE working-program.RUN
```

Restrictions on Writing PMA Modules for EPF Execution

When writing a module in PMA for execution within an EPF, several restrictions must be observed:

- Each subroutine in the module must execute in the V-mode or I-mode environment.
- If the module has impure procedure text, it must be declared as an impure module
- If the module has pure procedure text, it should be declared as a pure module
- Subroutines within the module must not use explicit addressing to externals unless their addresses are explicitly set during the BIND session
- Indirect Pointers (IPs) used in the module must never be modified by the module, because they are not necessarily reinitialized when the EPF is reinvoked

This section discusses these restrictions.

PMA Subroutines Must Execute in V-mode or I-mode Environment: A PMA subroutine intended for execution within EPFs must be assembled in the V-mode or I-mode environment, as implied by the requirements that PMA modules used for EPFs must begin with SEG or SEGR.

Under most circumstances, a PMA module must execute entirely in V-mode or I-mode. Occasionally, it may enter R-mode or S-mode to execute a limited set of instructions. For example, it may wish to execute a PIO instruction to read or test for a character from the user terminal. However, the PMA subroutine must reenter V-mode or I-mode before returning to the calling procedure.

Impure PMA Module Restrictions: If a PMA module is impure, the SEG or SEGR pseudo instruction at the top of the module must read SEG IMPURE or SEGR IMPURE.

An impure PMA module is characterized by an inability to be executed with the pure procedure (PROC) portion of the subroutine protected against modification by the subroutine. Instead, BIND places such a module in impure procedure (IMPURE) segments of an EPF. An IMPURE segment is similar to a PROC segment in that it contains procedure code and therefore must start at offset 0 in an actual segment, whereas DATA segments are relocatable to anywhere inside a segment. However, an IMPURE segment is not shared between users and is not protected against writing. Except in the case of a process-class library EPF, IMPURE segments are treated like DATA segments by PRIMOS, in that they are reinitialized each time the EPF is invoked.

Any PMA module that explicitly stores into the procedure text is inherently impure. Such modules are said to employ self-modifying code. This is widely regarded as poor programming practice. Moreover, some Prime systems employ preprocessors or a pipeline architecture, which may not behave as expected under such circumstances. On Prime systems, therefore, self-modifying code may not work or may result in nontransportable programs.

However, a PMA module can also implicitly modify procedure text, for example, by placing the ECB for the module in the procedure text and linking the module into an EPF. When such a module is part of an EPF, the actual placement of the linkage text is determined when the program is run, not when it is linked by BIND. (When loading with SEG and producing SEG runfiles, the linkage text is placed during the loading of the program by SEG.)

Therefore, when running as an EPF, PRIMOS must set the linkage base pointers for the ECB of each procedure in the EPF. If an ECB is in the procedure text, which is normally protected against writing, PRIMOS would encounter an access violation error if it tried to set the linkage base pointer for that ECB; therefore, PRIMOS does not attempt to modify the ECB. It is because the ECB requires modification at runtime that a module with an ECB in the procedure text is considered impure.

If BIND encounters an ECB in the procedure text, and the module is not declared as an impure module, BIND issues a warning message. If the resulting EPF is executed, it may produce an access violation error when the offending module is invoked, because the imaginary address has not been translated into an actual address.

Similarly, placing IPs (Indirect Pointers) in the procedure text results in an impure module when that module is linked using BIND, unless all such references identify external symbols that are explicitly located during program binding using the SYMBOL command of BIND.

Modification of procedure text can occur explicitly or implicitly. An explicit modification is performed by the subroutine code (or possibly code outside the subroutine). When the subroutine is run within an EPF, implicit modifications occur when subroutine linkage data are placed in the procedure text. This linkage information must be dynamically adjusted by the EPF mechanism when the subroutine is executed.

A JST (Jump and STore) instruction that references an internal subroutine also produces impure code, because JST stores the offset portion of the return address in the halfword that is the target of the instruction and then begins execution at the subsequent halfword. If the target of the JST instruction is in procedure code, rather than linkage, common, or stack frame storage, then the procedure code is impure. Instead, use the JSXB, JSX, or JSY instructions, and modify the target subroutine accordingly.

The RLIT and FIN pseudo instructions are often used to specify that literals are to be placed in the procedure text, rather than the linkage text. If literals are properly used, this does not result in an impure PMA module. However, using RLIT or FIN for literals that are to be stored into results in an impure module. (Storing into literals is considered extremely bad programming practice.) For example, the following literal reference is a pure reference independent of the use of RLIT or FIN:

```
LDA =5
```

However, the following literal reference requires that the RLIT or FIN pseudo instruction not be used if the procedure is to remain pure:

```
STA =10
```

This reference also has the dangerous side effect of causing references to the literal value of 10 to reference a different value for the entire subroutine or for portions of that subroutine.

Pure PMA Modules: If a PMA module is pure, that is, if it does not have any of the characteristics of an impure module as described above, then the SEG or SEGR pseudo instruction at the top of the module should read SEG PURE or SEGR PURE. If PURE is not specified, the default is PURE anyway.

However, explicitly including the PURE keyword can be a convenient signal to other programmers that the module has been checked for purity. If this convention is used, then any PMA module without a PURE or IMPURE keyword following the SEG or SEGR pseudo instruction should be checked for purity before being linked into an EPF.

No Explicit Addressing of Dynamically Placed Externals: If a PMA module attempts to use an explicit address to an external entity, and the external entity is not placed via the SYMBOL command during the BIND session, the PMA module may not execute properly.

Such an attempt might appear as follows:

```
            LDA THEVALUE,*
              .
              .
              .
THEVALUE    OCT 4001
            OCT 174000
```

To remedy this situation, either use the SYMBOL command to place the entity being addressed through THEVALUE at 4001/174000, or fix THEVALUE to appear as follows:

```
            EXT ENTITY
THEVALUE    IP  ENTITY
```

Do Not Store Into IPs or ECBs: If your program declares Indirect Pointers (IPs) or Entry Control Blocks (ECBs), it should never modify them during execution. For example, consider the following subroutine:

```
            SEG
            RLIT
            SYML
*
            ENT   TESTSUBR
*
            LINK
TESTSUBR    ECB   START
            PROC
*
START       CALL  TNOU
```

```
          AP    THE_IP,*S
          AP    =16,SL
*
          EAL   STRING2
          STL   THE_IP
*
          PRIN
*
STRING1   BCI   'THIS IS STRING 1'
STRING2   BCI   'THIS IS STRING 2'
*
          LINK
THE_IP    IP    STRING1
*
          END   TESTSUBR
```

This program uses THE_IP to point to one of two strings. It specifies that, initially, THE_IP is to point to STRING1, and that for all subsequent calls, THE_IP is to point to STRING2. The intention here is for the subroutine to behave differently during its first invocation by a program than it behaves during subsequent invocations by the program.

However, once THE_IP is modified, it is not reinitialized by PRIMOS during repeated invocations of the program unless the program has been removed from memory (or if the k$init_all key is supplied to EPF$INIT by a user program as described in Volume III of this series).

For example, if you call this subroutine from a program that simply calls TESTSUBR once and then exits, then the program does not produce identical results when invoked several times in a row, as shown in the following sample session:

```
OK, RESUME TESTPROG
THIS IS STRING 1
OK, RESUME TESTPROG
THIS IS STRING 2
OK, RESUME TESTPROG
THIS IS STRING 2
OK, REMOVE_EPF TESTPROG
OK, RESUME TESTPROG
THIS IS STRING 1
OK, RESUME TESTPROG
THIS IS STRING 2
OK,
```

The REMOVE_EPF command, used midway through this session, removed the EPF from memory. This forced the complete reinitialization of the EPF at the next RESUME command, and thus restored THE_IP to its initial state.

In any situation where you wish to modify IPs or ECBs, split them into:

- The desired initial value (IP or ECB) that is not modified by the program
- A block of linkage data (using the BSS pseudo instruction) that is to contain the actual value that is used and modified (BSS 2 for IP, BSS '20 for ECB) during program execution

Then, create another linkage-resident variable called FIRST_INVOCATION and declared as follows:

```
FIRST_INVOCATION OCT 1
```

Having done this, the first thing your subroutine should do is examine FIRST_INVOCATION. If nonzero, it should initialize the block of linkage data described above to the desired initial value (IP or ECB).

Then, before your subroutine returns, it should examine FIRST_INVOCATION again, and, if nonzero, it should update the block of linkage data as desired and then set FIRST_INVOCATION to 0.

Because FIRST_INVOCATION is an initialized datum, it is reinitialized by PRIMOS during every program invocation.

Here is the TESTSUBR subroutine, shown above, modified according to these recommendations:

```
            SEG
            RLIT
            SYML
*
            ENT   TESTSUBR
*
            LINK
TESTSUBR    ECB   START
            PROC
*
START       LDA   FIRST_INVOCATION
            BEQ   GO
*
            EAL   THE_IP_INITIAL,*
            STL   THE_IP
*
GO          CALL  TNOU
            AP    THE_IP,*S
            AP    =16,SL
*
            LDA   FIRST_INVOCATION
            BEQ   RETURN
*
```

```
          EAL   STRING2
          STL   THE_IP
*
          CRA
          STA   FIRST_INVOCATION
*
RETURN    PRTN
*
STRING1   BCI   'THIS IS STRING 1'
STRING2   BCI   'THIS IS STRING 2'
*
          LINK
THE_IP_INITIAL IP    STRING1
THE_IP    BSS   2
FIRST_INVOCATION OCT 1
*
          END   TESTSUBR
```

Now, invoking the TESTPROG program linked with the new version of TESTSUBR shown above produces the correct output during subsequent invocations:

```
OK, RESUME TESTPROG
THIS IS STRING 1
OK, RESUME TESTPROG
THIS IS STRING 1
OK, RESUME TESTPROG
THIS IS STRING 1
OK,
```

8
Shared Data

The architecture of Prime 50-Series systems allows programs to have access to memory that is shared by all processes on a system. Cooperating programs may use this capability to communicate between user processes.

In addition, programs may wish to communicate with each other within the context of a single process, by accessing a predefined common area in nonshared (per-user) memory.

The distinction here is between data that is shared for an entire system and data that is shared within a user process. Aside from the choice between system-wide and process-wide shared data, the mechanisms used in employing both kinds of shared data access are the same; exceptions are noted in this chapter.

This chapter describes:

- How to define a shared common area in either shared or nonshared memory
- How to update information in a shared common area atomically

HOW TO DEFINE A SHARED COMMON AREA

There are two general ways to define a common area that is shared between two or more programs, either within the context of one process (when nonshared memory is used) or of one system (when shared memory is used):

- Use the SYMBOL subcommand of BIND to specify the actual address of an external common area
- Place subroutines that wish to communicate with each other in a process-class library EPF

This section describes both approaches.

Using the SYMBOL Subcommand of BIND

The SYMBOL subcommand of BIND is used to specify the actual address of an external common area in memory. This address may be either in shared memory or in nonshared static memory.

Each program or library that wishes to use the shared area must use the SYMBOL subcommand during the linking of that program. In addition, they must all specify the same actual address of the shared data area. The name of the shared data area may differ from program to program; however, it is good practice to use the same name throughout.

Because the location of the shared common area is placed during the linking of a program, there are no special requirements on the compiling of the program. However, there are coding requirements that are described in the section later in this chapter entitled HOW TO UPDATE SHARED INFORMATION ATOMICALLY.

Before you can run a program that accesses a shared data area, you must:

1. Determine the address of the shared data area.
2. Ensure that the shared data is initialized once during each system coldstart (for data in shared memory) or once for each user login (for data in nonshared memory).
3. Specify the appropriate SYMBOL subcommand while linking your program or library.

Determining the Address of the Shared Data Area: Because memory accessed via the SYMBOL subcommand of BIND is not managed by PRIMOS, you must determine a location for your shared data area that does not conflict with other programs or libraries.

For shared (system-wide) memory, you use shared segments. Shared segment numbers range from '2000 through '2577 at Rev. 19.4. However, you must consult with your System Administrator before deciding what portions of shared system-wide memory to use. He or she will use the System Administrator's Guide, which contains a list of shared segment usage by Prime products, along with information on shared segment usage at your installation, to determine where your shared data can be placed.

For nonshared (per-user) memory, you use static segments. Static segment numbers range from '4000 up to the first dynamic segment number for each particular user. You may, for example, use memory in segment '4001. However, bear in mind that static-mode programs use segment '4000 and may use static segments beyond segment '4000. In addition, other users may be building programs which, like yours, use static segments to store shared per-user data; their choice of placement of the data might conflict with your choice. Again, it is best to consult your System Administrator on the matter. He or she may decide to keep a registry of static per-user segment allocation at your installation.

Ensuring That the Data Is Initialized: Before any program uses the shared data, the data must be initialized, either by a program that uses the data or by a special initialization program. The execution of a program, whether an EPF or a static-mode program, does not automatically cause common data placed via a SYMBOL subcommand to be initialized. This is because BIND cannot statically initialize a static segment, nor does PRIMOS perform any static segment initialization while preparing to execute an EPF.

For shared (system-wide) memory, this initialization must be performed at the supervisor terminal; typically, it is performed at system coldstart. The following sample command sequence may be entered interactively at the supervisor terminal or placed in the system startup file, PRIMOS.COMI (or C_PRMO):

```
OPRPRI 1  /* Allow SHARE commands.
SHARE 2030 700  /* Share segment 2030 for read and write access.
RESUME SYSTEM>INIT_MYPROG  /* Initialize segment 2030.
OPRPRI 0  /* Disallow SHARE commands.
```

Alternatively, if the initialization can be placed in a static-mode memory image, such as one generated via the SHARE subcommand of SEG, you can use a command sequence such as:

```
OPRPRI 1  /* Allow SHARE commands.
SHARE SYSTEM>MYPROG2030 2030 700  /* Share segment 2030 for read
      /* and write access, and load a static-mode memory image into
      /* it.
OPRPRI 0  /* Disallow SHARE commands.
```

In both cases, you must build either the initialization program or the initialization static-mode shared image.

Note

If you SHARE a new shared segment without specifying a static-mode memory image to be placed in it, the data in the segment is initialized to all zeroes. You may therefore actually consider not initializing the segment data at system coldstart, and choose instead to have your program initialize the data area if a special portion of it consists of zeroes (indicating that the data has not yet been initialized). If you do this, read the explanation below of how to perform initialization in this fashion, because this method is often used for nonshared (per-user) memory.

For nonshared (per-user) memory, the safest approach to initializing your data area is to ask all users who are going to use your program or library to modify their LOGIN.CPL or LOGIN.COMI file to include a command such as:

```
RESUME PROG_DIR>INIT_MYPROG  /* Initialize some static memory.
```

You must build the INIT_MYPROG program and place it in the appropriate directory (PROG_DIR in the example).

This method is the safest because it reduces the chances that your program will be run before the data is initialized.

An alternate method is for your program to determine, when it begins running, whether the data has been initialized, by testing a portion of the data area to see if it contains a certain data pattern. If it does not, your program can initialize it at that point in time, and set the data pattern to indicate that initialization has taken place.

This kind of dynamic initialization has the advantage of being easier to set up, but it has the disadvantage of possibly not recognizing a situation where the correct data pattern is in place but the entire data area has not, in fact, been initialized. To reduce this risk, have your initialization logic write the pattern only after the rest of the data area has been initialized, and use plenty of unusual values in the data pattern.

Specifying the Appropriate SYMBOL Subcommand: Specify the appropriate SYMBOL subcommand as follows:

```
SYMBOL name definition [size]
```

Here, name is the external name of the common area, definition is the location of the shared data area in the form segno/offset, and size, which is optional, specifies the size of the shared data area in 16-bit halfwords (in decimal, not in octal). For example, to specify that the external common area named MESSAGES is to be placed at address 2030/0 and that it is 200 decimal halfwords in length, issue the following subcommand while linking the program via BIND:

```
SYMBOL MESSAGES 2030/0 200
```

Using a Process-class Library EPF

If you want to place the shared data area in nonshared (per-user) memory, it might be best to place all subroutines that use the shared data area in a single process-class library EPF. The data area may then be either a common area or the linkage for a single procedure that contains one or more entrypoints. This approach has the following advantages over using the SYMBOL subcommand:

- BIND and PRIMOS ensure that the data area is automatically initialized each time the process-class library EPF is first invoked by a user.
- You do not need to check with your System Administrator to determine where, in static memory, to place the shared data area, because BIND and PRIMOS determine the location of the area when the process-class library EPF is first invoked by a user.
- You do not need to specify any special BIND subcommand aside from the commands to declare the library to be process-class.
- Additional memory may be allocated for the shared data area by using ALLOCATE statements in PL1/G.
- Data can be initialized by using STATIC INITIAL(...) attributes in PL1/G or DATA /.../ statements in FORTRAN.

However, you must ensure that the subroutines you place in the process-class library EPF meet the requirements for a process-class library EPF, as outlined in Chapter 6.

A small but illustrative example of this form of shared data access is the GET_USERNAME subroutine shown in Chapter 6 and reproduced here:

```
get_username: proc returns(char(32));

dcl 1 timdat_info static,
      2 date char(6),
      2 time fixed bin(15),
      2 ticks fixed bin(15),
```

```
        2 meters (4) fixed bin(15),
        2 tps fixed bin(15),
        2 user_number fixed bin(15),
        2 user_name char(32);

    dcl have_info bit(1) static init('0'b);

    dcl timdat entry(1,2 char(6),2 fixed bin(15),2 fixed bin(15),
              2 (4) fixed bin(15),2 fixed bin(15),2 fixed bin(15),
              2 char(32),fixed bin(15));

    if ^have_info
       then do;
            call timdat(timdat_info,28);
            have_info='1'b;
            end;

    return(user_name);
    end;
```

Because the task of this subroutine is simply to return the username of the invoking user, it is written to save time after the first invocation by calling the TIMDAT subroutine, which is an internal PRIMOS entrypoint, only once. After the first call, subsequent invocations result simply in the returning of the username acquired during the first invocation.

If GET_USERNAME is placed in a process-class library EPF, then several programs that call it may be run, and yet it will invoke TIMDAT only the first time it is run. PRIMOS keeps the linkage data for a process-class library EPF active between program invocations, as described in Chapter 6.

In a sense, this subroutine "communicates" between program invocations within a single user process. One program may have to end up calling TIMDAT via GET_USERNAME to acquire the username of the user, but the next program run by that user reuses the same username. This is a very simple form of interprogram communication.

This simple use of process-class library EPF linkage is applicable to information that is process-wide, rather than program-wide. Other examples of such information include:

- User number
- Terminal type — the user may have to be asked to enter the terminal type the first time, but after that, it could be made available to subsequent programs and libraries via a process-class library EPF subroutine
- User's origin directory

- User's erase and kill characters (useful if the process-class library EPF also handles the collection of command lines; the internal PRIMOS subroutine ERKL$$ is already quite fast)

More complex uses of process-class library EPF linkage (or common) data areas might include:

- Keeping track of virtual circuits a user has open to other systems for remote login or other purposes, allowing a user to leave a networking program and later reenter it while keeping his or her connections intact

- Keeping track of how many times a user invoked one or more particular applications and possibly even keeping track of functions invoked within each application — a log file unit number might be a candidate for a shared datum

However, except for the very simplest uses of shared per-user data areas, you must be careful to design subroutines so that they either prevent interruption (such as by inhibiting quits or establishing on-units) or are robust enough to withstand interruption at any point followed by a new invocation of the same or similar subroutines. Remember, a subsequent invocation of an executing subroutine in a process-class library EPF, while recursive in nature, never causes the allocation or initialization of linkage data; the existing linkage data is used. Modifications made by the second invocation of the subroutine can affect the ability of the first invocation of the subroutine to complete correctly when its execution is resumed. See the next section for information on how to update shared information in a manner that protects against multiple concurrent updates.

HOW TO UPDATE SHARED INFORMATION ATOMICALLY

Whether in system-wide shared memory or in per-user nonshared memory, a data area must be protected against the possibility of multiple concurrent updates if:

- Updates can be performed by two processes — only for system-wide shared memory

- Updates can be performed by either two or more separate subroutines or two separate invocations of a single subroutine. This occurs when one of the subroutines can be invoked following an interruption of one of the subroutines (such as via a user typing CONTROL-P or the asynchronous invocation of an on-unit as a result of a condition such as PH_LOGO$ or LOGOUT$ being signaled)

Incorrect or nonexistent use of interlocking facilities for the updating of shared data areas is very difficult to detect; similarly, correct use is difficult to prove. Typically, incorrect or nonexistent use of such facilities results in perhaps a serious program error once

in a great while; because the serious error is not easily reproducible, it is often written off as a "glitch". To build the most robust product, you should carefully analyze portions of your product in which interlocking facilities might be needed to determine just how to use them.

You should study how your subroutines use either the linkage area of a process-class library EPF or, more obviously, shared system-wide memory to see if interrupt inhibition or atomic updating are called for. Keep in mind that statements such as

```
ITEMS=ITEMS+1
```

are interruptible between the reading of the current value of ITEMS and the storing of the incremented value. An interruption at that point, followed by reinvocation of the same subroutine, might result in two invocations of the subroutine, causing the ITEMS value to be incremented only once. If ITEMS is used as a unique identifier by subroutine invocations, the results might be disastrous; yet this situation might not ever occur even during exhaustive testing, and would probably not be reproducible. On the other hand, if the increment of ITEMS is tested against some other static data in an atomic fashion, perhaps the failure of the multiple concurrent update will not have a bad effect on the program.

For example, see the final version of the AVERAGE subroutine in Chapter 6. Note that while it makes reasonably heavy use of linkage (common) data, only certain updates to the common areas are protected by calling a special subroutine. Other updates are interruptable because they are protected by the atomic updates performed, ultimately, by the STAC instruction in the COND_STORE subroutine.

Following is a listing of the COND_STORE subroutine, written in PMA, plus a version of the subroutine named LONG_COND_STORE that deals with FIXED BIN(31) (FULL INT, or INTEGER*4) integers. After that, a listing of a special-purpose PMA subroutine that performs a typical use of the STAC instruction, named INCREMENT, is provided.

Note

These three subroutines, COND_STORE, LONG_COND_STORE, and INCREMENT, are provided by Prime only in this document, not on the master disk. If you wish to use them, you must key them into your system. They are all PMA subroutines.

The COND_STORE.PMA Subroutine

The COND_STORE.PMA subroutine is called with three halfword integers, and returns as its function value a halfword integer. In PL1/G, its calling sequence is:

```
dcl cond_store entry(fixed bin(15),fixed bin(15),fixed bin(15))
          returns(fixed bin(15));

cond_store_ok=cond_store(destination,new_value,old_value);
```

If cond_store_ok is 1, then the value of destination has been successfully changed from old_value to new_value. Otherwise, cond_store_ok is 0, and no change to destination has taken place.

```
          SEG PURE
          RLIT
          SYML
*
          SUBR COND_STORE,ECB
*
          LINK
ECB       ECB COND_STORE,,WHERE,3
          PROC
*
COND_STORE EQU *
          ARGT
          LDA OLD,*
          TAB
          LDA NEW,*
          STAC WHERE,*
          BCEQ OK
          CRA
          PRTN
*
OK        LT
          PRTN
*
          DYNM WHERE(3),NEW(3),OLD(3)
*
          END
```

The LONG_COND_STORE.PMA Subroutine

The LONG_COND_STORE.PMA subroutine is called with three fullword integers, and returns as its function value a halfword integer. In PL1/G, its calling sequence is:

```
dcl long_cond_store entry(fixed bin(31),fixed bin(31),
          fixed bin(31)) returns(fixed bin(15));

cond_store_ok=long_cond_store(destination,new_value,old_value);
```

If cond_store_ok is 1, then the value of destination has been successfully changed from old_value to new_value. Otherwise, cond_store_ok is 0, and no change to destination has taken place.

```
          SEG PURE
          RLIT
          SYML
*
          SUBR LONG_COND_STORE,ECB
*
          LINK
ECB       ECB LONG_COND_STORE,,WHERE,3
          PROC
*
LONG_COND_STORE EQU *
          ARGT
          LDL OLD,*
          ILE
          LDL NEW,*
          STLC WHERE,*
          BCEQ OK
          CRA
          PRTN
*
OK        LT
          PRTN
*
          DYNM WHERE(3),NEW(3),OLD(3)
*
          END
```

The INCREMENT.PMA Subroutine

The INCREMENT.PMA subroutine increments the value of a variable atomically and returns the new value as its function value. It deals entirely in halfword integers (FIXED BIN(15), INTEGER*2, HALF INT). Note that by the time the subroutine actually returns, the returned value may differ from the latest value of the variable due to an update

by another procedure or process; the purpose of the returned value is to provide the caller with a value guaranteed to be unique if all references to the variable by all procedures and processes are done through the INCREMENT subroutine.

```
          SEG PURE
          RLIT
          SYML
*
          SUBR INCREMENT,ECB
*
          LINK
ECB       ECB INCREMENT,,VARIABLE,1
          PROC
*
INCREMENT EQU *
          ARGT
*
TRY_AGAIN LDA VARIABLE,*
          TAB
          AlA
          STAC VARIABLE,*
          BCNE TRY_AGAIN
*
* It is important to leave A-reg as is after the STAC
* instruction succeeds; do not LDA VARIABLE,* or it may
* be different from what we incremented it to just now!
*
          PRTN                          New value in A-reg
*
          DYNM VARIABLE(3)
*
          END
```

9
Maps and Addresses

Because EPFs are dynamically placed in available memory at runtime, maps of EPFs produced by BIND do not, for the most part, contain actual memory addresses. Instead, they contain imaginary addresses, as described in Chapter 1.

Once an EPF has been mapped to memory, the location of its procedure code is determined; once that EPF has had its linkage data allocated, the location of all data in the EPF is determined. To display information on where code and data for an active or mapped EPF have been placed, use the following format of the LIST_EPF command:

```
LIST_EPF EPF-name -SEGMENTS
```

You may then correlate the displayed output from the LIST_EPF command with the BIND map produced for that EPF to determine the actual addresses of subroutines, common areas, and so on, in that particular invocation of the EPF.

This chapter explains imaginary versus actual addresses. Then, this chapter describes how to use LIST_EPF output, LIST_SEGMENT output, BIND maps, VPSD, and DUMP_STACK output to examine in-memory EPFs. Finally, the chapter provides a short section on expanded listings.

IMAGINARY VS. ACTUAL ADDRESSES

To understand how to correlate BIND maps with the displayed output of PRIMOS commands such as LIST_EPF, DUMP_STACK, and VPSD, you must first understand imaginary addresses and how they differ from actual addresses.

An imaginary address is a temporary representation of a memory address generated by BIND. An imaginary address identifies locations within an EPF for later correlation with actual addresses determined by PRIMOS when an EPF is mapped, its linkage allocated, and its linkage initialized. An actual address is also known, from a system architecture point of view, as a virtual memory address.

Both imaginary addresses and actual addresses have the form:

segno/offset

So that imaginary addresses and actual addresses can be distinguished at a glance, imaginary addresses have signed segment numbers, while actual addresses have unsigned segment numbers. Here are some sample addresses:

Imaginary Addresses	Actual Addresses
-0002/10472	4376/15433
+0000/77160	4363/126021
-0004/1672	4232/1000
+0006/1000	4337/100123

(The addresses listed above do not necessarily have any correlations with each other.)

Sometimes, actual addresses are shown with ring numbers, as in 4376(3)/15433 or 4337(0)/100123. These ring numbers have no effect on the execution or behavior of your program; therefore, you may ignore them.

Also, different portions of PRIMOS and BIND display both types of address with or without leading zeroes. Therefore, imaginary address -0002/10472 may also be displayed as -2/10472 or as -0002/010472; all three displays are equivalent.

Positive or Negative Segment Numbers

Imaginary addresses have signed segment numbers; either a + or - always precedes the segment number of an imaginary number.

- A positive (+) sign indicates a segment used to hold pure procedure code.
- A negative (-) sign indicates a segment used to hold linkage or common data, or impure procedure code.

In addition, BIND maps indicate the type of information stored in each segment. In BIND maps, pure procedure segments are labeled PROC; data segments containing linkage or common data are labeled DATA; and impure procedure code segments are labeled IMPURE. PROC segment numbers always start with a positive (+) sign; DATA and IMPURE segment numbers always start with a negative (-) sign.

Thus, imaginary segment number +0 is a PROC (pure procedure) segment, whereas imaginary segment number -2 is an impure segment that may be either a DATA or an IMPURE segment.

USING THE LIST_EPF COMMAND

The LIST_EPF command is the crucial command to use when working with EPFs and memory addresses. It displays the correspondence between imaginary segment numbers and actual memory addresses.

For example, here is the display of a LIST_EPF -SEGMENTS command issued after the LD, COPY, and DELETE commands have been used:

```
OK, LIST_EPF -SEGMENTS

1 Process-Class Library EPF.

(active)      <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
  2 procedure segments:   +0:4234              +2:4235
  2 linkage areas:        -2:4376(0)/0         -4:4377(3)/730

3 Program EPFs.

(not active) <SYSDSK>CMDNC0>COPY.RUN
  1 procedure segment:    +0:4237
  2 linkage areas:        -2:4375(0)/0         -4:4377(3)/44744
(not active) <SYSDSK>CMDNC0>DELETE.RUN
  1 procedure segment:    +0:4240
  2 linkage areas:        -2:4374(0)/0         -4:4377(3)/52770
(not active) <SYSDSK>CMDNC0>LD.RUN
  1 procedure segment:    +0:4236
  1 linkage area:         -2:4377(3)/36100

OK,
```

The correspondence between imaginary PROC segment numbers and actual addresses is displayed as:

```
+imaginary_segno:actual_segno
```

The correspondence between imaginary DATA and IMPURE segment numbers and actual addresses is displayed as:

```
-imaginary_segno:actual_segno/actual_offset
```

Notice how each PROC segment (with RX access) has only one EPF listed for it in the LIST_SEGMENT display; whereas segment 4377, a DATA segment, has four EPFs listed for it. This is because an imaginary PROC or IMPURE segment must be the only resident of an actual segment, even if it does not use all of it; imaginary segments containing executable procedure code must be placed at offset 0 of an actual segment because executable procedure code is not relocatable within a segment. However, imaginary DATA segments can share actual segments with other imaginary DATA segments, because they can be easily relocated within a segment.

Therefore, in the output from the LIST_EPF -SEGMENTS command shown earlier in this section, the actual addresses for imaginary PROC segments are shown without offset portions, as in:

```
+0:4240                                          (a PROC segment)
```

The actual addresses for imaginary DATA segments are shown with nonzero offset portions, as in:

```
-4:4377(3)/52770                                 (a DATA segment)
```

For EPFs containing imaginary IMPURE segments, LIST_EPF -SEGMENTS displays the actual addresses with zero offset portions, as in:

```
-2:4374(0)/0                                     (an IMPURE segment)
```

Note

The LIST_EPF -SEGMENTS command displays information on only the most recent invocation of an EPF. Previous active invocations, applicable for program EPFs and program-class library EPFs, are not displayed in the list of linkage segments.

USING THE LIST_SEGMENT COMMAND

To illustrate the actual mapping of segments for the EPFs displayed in the sample LIST_EPF display above, here is the display from a LIST_SEGMENT -NAME command issued just after the same LIST_EPF -SEGMENTS command shown above:

```
1 Private static segment.
segment access
-------------
4000    RWX

9 Private dynamic segments.
segment access  epf
-------------------
4234    RX      <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
4235    RX      <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
4236    RX      <SYSDSK>CMDNC0>LD.RUN
4237    RX      <SYSDSK>CMDNC0>COPY.RUN
4240    RX      <SYSDSK>CMDNC0>DELETE.RUN
4374    RWX     <SYSDSK>CMDNC0>DELETE.RUN
4375    RWX     <SYSDSK>CMDNC0>COPY.RUN
4376    RWX     <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
4377    RWX     <SYSDSK>CMDNC0>COPY.RUN
                <SYSDSK>CMDNC0>DELETE.RUN
                <SYSDSK>CMDNC0>LD.RUN
                <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN

OK,
```

The segments with RX access are PROC segments — the R indicates that they may be read, while the absence of the W indicates that they cannot be written. The segments with access RWX can be both read and written; they are either DATA or IMPURE segments. As explained in Chapter 1, PRIMOS automatically shares PROC segments between users using the same EPF.

USING THE BIND MAP

The map produced by a MAP subcommand while in BIND shows the relationship of locations internal to an EPF to the beginning of two or more imaginary segments that are used by that EPF. Each EPF has at least two segments: a PROC segment and a DATA segment. Larger EPFs may use more than one PROC or DATA segment each, and some EPFs have IMPURE segments (which are like PROC segments except they are modified during execution and hence cannot be shared or protected against modification).

The BIND map primarily contains imaginary addresses. The exceptions occur when the SYMBOL subcommand is used to specify an actual address

for an external symbol (such as a common block). In this case, the map shows an actual address (with an unsigned segment number) for that symbol, and the list of segments at the top of the map show that segment as a STATIC type segment.

Once you have begun executing an EPF, its imaginary addresses have been resolved to actual addresses. You can then use the LIST_EPF -SEGMENTS command to display the mapping from imaginary segment numbers in the BIND map to actual addresses in user memory. You can then add the offset portion of a symbol's imaginary address to its corresponding actual address in the LIST_EPF display to determine the symbol's actual address in user memory.

For example, suppose a line in a BIND map reads:

```
Name       ECB address   Initial PB%  . . .  Initial LB%
SUBR1    -0002/000004  +0000/001005  . . .  -0002/177400
```

Also, suppose a LIST_EPF -SEGMENTS command displays the following information for the EPF containing SUBR1:

```
(active) <USRDSK>UNGER>MY_PROG.RUN
  1 procedure segment:     +0:4234
  1 linkage area:          -2:4377(3)/730
```

Determining Procedure Code Addresses

To determine the address of the first instruction of SUBR1, you first examine the imaginary address in the column labeled Initial PB%. Here, the imaginary address is +0000/001005. Next, you look up imaginary segment +0 in the LIST_EPF display to determine its corresponding segment number. Here, +0 corresponds to segment 4234. Therefore, the actual address for the first executable instruction in SUBR1 is 4234/1005.

Determining ECB Addresses

To determine the address of the ECB of SUBR1, you examine the imaginary address in the column labeled ECB address. Here, the imaginary address is -0002/000004. Next, you look up imaginary segment -2 in the LIST_EPF display to determine its corresponding segment number. Here, -2 corresponds to actual address 4377/730. You add 730 to 4 (without ever carrying into the segment number portion) to determine the actual address of the ECB for SUBR1, 4377/734.

Determining Stack Frame Addresses

To determine the address of a stack frame for an active procedure, use the DUMP_STACK command, described later in this chapter.

Determining Addresses For Other Map Objects

To determine the address of other objects in a BIND map, such as common blocks, dynamic links, and so on, simply use the same method used for determining procedure code addresses (when a positive imaginary segment number is involved) or for determining ECB addresses (when a negative imaginary segment number is involved).

However, for link frame addresses, read the next section carefully before proceeding.

Determining Link Frame Addresses

Finally, let's look at the trickiest calculation. To determine the address of the link frame of SUBR1, you examine the imaginary address in the column labeled Initial LB%. Here, the imaginary address is -0002/177400. However, this is not the imaginary address of the link frame itself; it is, as implied by the column header, the address placed in the Link Base (LB) when the procedure is called. On Prime systems, the address in the LB is always the starting address of the link frame minus '400 (in halfwords, without borrowing from the segment number portion).

Therefore, when considering the link frame, you must decide whether you wish to determine the address of the beginning of the link frame itself or the address placed in the LB when the procedure is executed. The address of the link frame is useful when you want to examine the link frame data. The address placed in the LB is useful when you want to examine procedure code that uses the link frame, because assembler instructions include the '400 halfword offset; the instruction LDA LB%+'400 references the first halfword in the link frame.

In fact, determining both values is straightforward. As before, imaginary segment -2 corresponds to address 4377/730, so you add 730 to 177400 (without carrying) to determine the address placed in the LB for the SUBR1 procedure, 4377/330. To determine the actual address of the beginning of the link frame for a procedure, add '400 to the LB for the procedure (again, without carrying into the segment number). Here, the beginning of the link frame for SUBR1 is 4377/730.

PRIMOS commands that display memory allocation information, such as LIST_EPF, display link frame addresses. PRIMOS commands that display base register and instruction information, such as DUMP_STACK, VPSD, and PM, display LB addresses, which are '400 less than the actual locations of corresponding link frames.

USING VPSD

When you use VPSD (Virtual Prime Symbolic Debugger), the arithmetic is handled by the computer. The method for accessing a particular location in memory, given its imaginary address and the actual addressing corresponding to the imaginary segment number, is:

```
SN actual-segment-number
RE actual-offset
A >imaginary-offset
```

In VPSD, the SN subcommand sets the segment number; the RE subcommand sets the relocatable address offset, which is particularly useful for this sort of job; the A subcommand accesses (opens) a memory location, and the > specifies that the value following the > is to be treated as relative to the value specified in the RE subcommand.

For example, to access the ECB for SUBR1, which is at imaginary address -0002/000004 and where imaginary segment -2 corresponds to actual address 4377/730, you would issue the following VPSD subcommands:

```
SN 4377
RE 730
A >4
```

For addresses in PROC or IMPURE segments, which always have an actual-offset of 0, the method is even simpler:

```
SN actual-segment-number
A imaginary-offset
```

For example, to access the first executable instruction in SUBR1, which begins at imaginary address +0000/001005, and where imaginary segment +0 corresponds to actual segment 4234, you would issue these VPSD subcommands:

```
SN 4234
A 1005
```

To set up the LB register to access the link frame for a particular procedure, given the imaginary address from the Initial LB% column of the BIND map and the corresponding actual segment number from the LIST_EPF command display, use the following method:

```
LB actual-segment-number actual-offset+imaginary-offset
```

The LB subcommand of VPSD sets up an internal representation of the LB register, used when you make LB-relative references in VPSD (such as by typing something like A LB%+10). Three other similar commands, PB, SB, and XB, set up the internal copies of the other three base registers for the same purpose. (The + is underlined to distinguish it from dashes in the variable names; you type in the + as shown, but you substitute actual values for the variable names.)

For example, to set up the LB for the SUBR1 procedure, whose imaginary LB address is -0002/177400 and where imaginary segment -2 corresponds to actual address 4377/730, you would issue these VPSD subcommands:

```
LB 4377 730+177400
```

You would then reference the first halfword of the link frame of SUBR1 by typing:

```
A LB%+400
```

USING THE DUMP_STACK COMMAND

The DUMP_STACK command is necessary for displaying the addresses of stack frames in your program. Because it displays all of the stack frames between the latest frame and the initial frame at command level 1, identification of which frame is the desired frame is not necessarily straightforward. There are two ways to identify the proper stack frame:

- By the Owner= label, if the procedure you are attempting to locate is written in PL1/G, F77, VRPG, or Pascal
- By comparing the (LB=) field to the LIST_EPF -SEGMENTS display, and then to the BIND map of the appropriate EPF, which works for a procedure written in any language

The first method requires that the procedure identify itself by setting up its stack frame header so that it identifies its ECB, and by having the ECB contain the name of the procedure. However, not all Prime language translators generate code to perform this action each time a procedure is invoked; for example, FTN and PMA do not perform the requisite actions. In such cases, you must compare LB values.

Typically, you are searching for the stack frame of a particular procedure. In this case, determine the Initial LB% for the procedure according to the instructions given earlier in this chapter. Then, look for the corresponding stack frame in a DUMP_STACK display. If it isn't there, it means that the procedure you are searching for is not active and therefore not on the stack.

How to Locate the Stack Frame for a Procedure

Here is a sample display from the DUMP_STACK command after a user interrupted a running program EPF by typing CONTROL-P:

```
OK, DUMP_STACK
Backward trace of stack from frame 7 at 6002(3)/4046.

STACK SEGMENT IS 6002.

(7) 004046: CONDITION FRAME for "QUIT$"; returns to 13(3)/77622.
    Condition raised at 4257(3)/1327; LB= 4377(0)/44424, Keys= 004000

(8) 003726: FAULT FRAME; fault type "RXM" (0)
    Fault returns to 4257(3)/1327; LB= 4377(0)/44424, keys= 004000
    Fault code= 000000, fault addr= 4257(3)/130004.
    Registers at time of fault:
                  Save Mask= 007755;  XB= 4257(3)/1042
      GR0  60013  24667 14002624667   GR1      0      0           0
    L,GR2      0     12          12 E,GR3    310  11766    62011766
      GR4      0      0           0 Y,GR5      0 174052      174052
      GR6   3466  66002   715466002 X,GR7      0   1061        1061
     FAR0 4375(3)/12         FLR0          13 FR0    2.41041274E-39
     FAR1 4377(3)/124        FLR1     3466002 FR1    3.75089893E 543

(9) 003710: Owner=  (LB= 4377(0)/44424).
    Called from 4234(3)/1055; returns to 4234(3)/1061.

(10) 003616: Owner= SUBR1 (LB= 4377(0)/177460).
    Called from 41(3)/125336; returns to 41(3)/125340.

(11) 003462: Owner=  (LB= 41(0)/125010).
    Called from 13(3)/17354; returns to 13(3)/17376.
    Proceed to this activation is prohibited.

(12) 002220: Owner=  (LB= 13(0)/20256).
    Called from 13(3)/15025; returns to 13(3)/15033.

(13) 001420: Owner=  (LB= 13(0)/20256).
    Called from 13(3)/7224; returns to 13(3)/7236.

(14) 000640: Owner=  (LB= 13(0)/11206).
    Called from 13(3)/163464; returns to 13(3)/163470.

(15) 000632: Owner=  (LB= 13(0)/163102).
    Called from 4(0)/163466; returns to 4(0)/0.
OK,
```

In this example, the stack frames for the user's program are numbers (9) and (10). They are easily distinguished because the LB segment numbers are in the private per-user segment range (4377 in this example), rather than in public shared PRIMOS segments (41 and 13 in

this example). In addition, Frame (10) is easily identified as belonging to a procedure named SUBR1 because it is a PL1/G procedure that identifies itself by name. The stack frame for SUBR1 is 6002/3616, where 6002 is the stack root (as displayed at the beginning of the DUMP_STACK display); the stack frame for the PMA subroutine it called, from which the QUIT$ condition was signaled via a user typing CONTROL-P, is 6002/3710.

However, in more complicated situations, the stack frames are often not so easy to identify, so comparison against the LB registers displayed for each frame are helpful.

Multiple Entrypoints With the Same LB

In a situation where more than one entrypoint has the same LB, identification by LB is insufficient. Here, identification by ECB is required. To do this, examine the stack frame to determine the address of the calling instruction. Then, examine the next higher-numbered stack frame to determine the contents of the SB and LB registers for the calling procedure. Then, enter VPSD and examine the calling instruction to determine the address of the ECB. Often, the calling instruction is a PCL instruction that goes indirect through an IP in the link frame (LB-relative); if this is the case, you must also set up the LB in VPSD to be the LB for the calling procedure. Occasionally, the calling instruction is a PCL through an SB-relative IP, in which case you must set up the SB in VPSD accordingly.

However, if the PCL is XB-relative, tracking down the actual address can be very difficult because the XB register contents can be changed during the processing of argument templates (APs) and its contents at the time of the call are not saved by the PCL mechanism. In this case, your best bet is to backtrack through the code prior to the PCL to determine how it calculated the address for XB by looking for an EAXB that is SB-, PB-, or LB-relative, and then reconstruct the sequence of instructions to determine the actual XB contents used at the time of the call.

Example: For example, to determine the address of the ECB that corresponds to the SUBR1 procedure, Frame (10) in the above example, we first examine the display for Frame (10):

```
(10) 003616: Owner= SUBR1 (LB= 4377(0)/177460).
     Called from 41(3)/125336; returns to 41(3)/125340.
```

This tells us that the instruction that called the SUBR1 procedure is at address 41/125336.

Next, we look at the next frame, Frame (11):

```
(11) 003462: Owner=  (LB= 41(0)/125010).
     Called from 13(3)/17354; returns to 13(3)/17376.
```

This tells us that the SB for the calling procedure is 6002/3462 and the LB for it is 41/125010. Now, we enter VPSD, examine the calling instruction, and track down its IP to the ECB. Once we have the actual ECB address, we use LIST_SEGMENT to show us which EPFs have linkage data in that segment; we then use LIST_EPF -SEGMENTS on the most likely EPF in the LIST_SEGMENT display to determine whether there is a matchup between the ECB address, the actual address of the linkage data for the EPF, and the imaginary ECB address in the BIND map.

```
OK, VPSD

$SN 41

$A 125336:S
41/ 125336 PCL% SB%+ 107,*    /
$SB 6002 3462

$A SB%+107:O
6002/ 3571 4377 (CR)
6002/ 3572 7050 /
$Q
OK, LIST_SEGMENT 4377 -NAME

1 Private dynamic segment.
segment access  epf
----------------

4377    RWX     <USRDSK>UNGER>MYLIBRARY.RUN
                <SYSDSK>LIBRARIES*>SYSTEM_LIBRARY.RUN
                <USRDSK>UNGER>MY_PROG.RUN

OK, LIST_EPF MY_PROG -SEGMENTS

1 Program EPF.

(active)      <USRDSK>UNGER>MY_PROG.RUN
  1 procedure segment:     +0:4235
  1 linkage area:          -2:4377(3)/7044

OK,
```

The difference between an actual ECB address of 4377/7050 and an actual linkage data address of 4377/7044 is 4, yielding a corresponding imaginary ECB address of -2/4, or -0002/000004 as shown in the examples of SUBR1 earlier in this chapter.

Examining the Stack Frame for a Procedure Invocation

Once you know the address of a stack frame for a particular procedure invocation, you can reenter VPSD and examine the stack frame by issuing the following commands:

```
VPSD
SB stackroot offset
A SB%+n
```

Here, stackroot is the segment number of the stack root, displayed by DUMP_STACK at the top of its display and subsequently during the display if the stack switches to another segment. (Watch for the STACK SEGMENT IS messages during the display; use the most recent message displayed before the target stack frame.)

The offset value comes from the octal number following the stack frame number. In the above DUMP_STACK example, the offset value for Frame (10) is 3616.

USING EXPANDED LISTINGS

Prime language translators all have the ability to produce expanded listings. An expanded listing is a special type of listing, obtained by including the -EXPLIST option on the compiler command line, that shows the partially processed machine instructions generated as a result of compiling each line in the source file, along with the offset of each instruction from the beginning of the program. However, there are several different formats of expanded listing output, and the offset values differ in meaning, such as between FTN and PL1G. Consult the language manual appropriate for your compiler for more information.

The listings produced by PMA correspond, of course, to instructions in the source program. The offsets shown are from the beginning of the procedure code, the link frame, the stack frame, and so on.

10
Binary Editors

This chapter describes the Binary Editor (EDB) and the Library Editor (LIBEDB). EDB is used to create and modify object (.BIN) library files. LIBEDB is used once a library is created to decrease linking or loading time. Both of these programs operate on object text blocks generated by Prime language translators such as F77, FTN, CBL, PL1G, PMA, and so on. These object-text blocks form the input to BIND, LOAD and SEG. The term linker is used to identify all three programs.

LIBEDB

The LIBEDB program is used for editing bypass information into library files. The linker uses the bypass information to skip an unnecessary routine efficiently instead of reading and discarding all the unwanted object text. Depending on the size and number of unnecessary routines in a library, the linker may process library files up to 50 percent faster if they have first been processed by LIBEDB.

LIBEDB is maintained as the runfile LIBEDB.SAVE in the UFD LIB. It should be used on a library file after its creation and after each time that the library is edited with the Binary Editor. The linker is capable, however, of handling a library which is not, or is only partially, processed by LIBEDB.

Because it is expected that LIBEDB will be used fairly infrequently, the user/computer interaction is self-explanatory. LIBEDB asks for an input and output filename and for file type. In theory, a library with large routines will link faster if it is created as a Direct Access Method (DAM) file. In practice, none of the regularly used libraries contain routines large enough to warrant creating the library as a DAM file instead of as a Sequential Access Method (SAM) file.

EDB

The command format for EDB is:

```
EDB { input-file } [output-file]
    {   -ASR    }
    {   -PTR    }
```

Both the input and output file may be pathnames. The input file should be an existing library or the binary output of a Prime language translator. The output file is optional; if specified, a file of that name is created if none exists. -ASR or -PTR instead of a file on the command line specifies a user terminal or paper-tape reader/punch, respectively. If these are not included, a PRIMOS file is assumed. (-ASR and -PTR are tremendously obsolete options.)

EDB displays ENTER, and then waits for user commands.

Operation

EDB maintains a pointer to the input file. When EDB is initialized, or after a TOP or NEWINF subcommand, the pointer is at the top of the input file. The pointer can be moved by the FIND subcommand to the start of a module. A module is identified by its subprogram or entry-point name. After a COPY subcommand (which copies blocks from the input to output file), the pointer is positioned to the module following the module copied.

Subcommand Summary

EDB responds to the following subcommands, listed in alphabetical order. Subcommands may be abbreviated to the underlined letters.

Note

The keyword ALL, used in the COPY and FIND subcommands, is not specially treated by EDB; if the external symbol name ALL is encountered in the input file, the COPY or FIND operation is terminated. This distinction is important only for input files that contain an external symbol name of ALL; in such a case, use some random name instead of ALL to COPY or FIND all modules in an input file, such as FDSA. The ALL keyword is essentially an ad hoc standard.

▶ BRIEF

Inhibits the display of subroutine names and entrypoints as they are encountered in the input file. (See also TERSE and VERIFY.)

▶ COPY { name | ALL | <RFL> | <SFL> }

Copies to the output file all main programs and subroutines from the pointer up to (but not including) the subroutine called name or containing name as an entrypoint. If name is not encountered or if COPY ALL is specified, EDB copies to the end of the input file and displays .BOTTOM. on the terminal. EDB moves the pointer past the last copied item.

<RFL> and <SFL> are special keywords that search for a reset-force-load or set-force-load flag block.

▶ FIND { name | ALL | <RFL> | <SFL> }

Moves the pointer up to the module of the input file containing a subroutine called name or containing name as an entrypoint without copying the intervening modules to the output file. If name is not found, EDB moves the pointer to the end of the input file and displays .BOTTOM. on the terminal.

In VERIFY mode, the FIND ALL command is useful for displaying all subroutines and entry names in the input file.

<RFL> and <SFL> are special keywords that search for a reset-force-load or set-force-load flag block.

▶ INSERT pathname

Copies all modules of pathname to the output file. The pointer to the original input file is unchanged.

▶ NEWINF pathname

Closes the current input file and opens pathname as the new input file. The pointer is positioned to the beginning of pathname.

▶ OPEN

Closes the current output file and opens pathname as the new output file.

▶ QUIT

Closes all files and exits to PRIMOS.

▶ REPLAC name pathname

Replaces the object module containing name as an entrypoint by all modules of pathname.

▶ RFL

Writes a reset-force-load flag block to the output file. Typically, all libraries begin with an RFL. The RFL block places a linker in library mode; while in library mode, only those modules that are referenced are linked. RFL mode is in effect until the linker encounters an SFL block.

Note

Because an RFL block affects other files linked after the object file containing the RFL block, it is important that any object file containing an RFL block contain an SFL block at the end of the file. See the SFL command.

▶ SFL

Writes a set-force-load flag block to the output file. This block places a linker in force-load mode; all subsequent modules are linked, whether or not they are called. SFL mode is in effect until the linker encounters an RFL block. A library file should be terminated by an SFL block.

▶ TERSE

Places the editor into TERSE mode. While in TERSE mode, EDB displays only the first entrypoint name of each module encountered. (See also BRIEF and VERIFY.)

▶ TOP

Moves the pointer to the top of the input file.

▶ VERIFY

Places EDB into VERIFY mode. All subroutine names and entrypoints, as they are encountered by EDB, are displayed on the terminal. EDB is initialized in the VERIFY mode. (See also BRIEF and TERSE.)

Obsolete Commands

The following commands are outmoded but are included for the sake of compatibility:

▶ ET

Writes an end-of-tape mark on the output file ('223, '223 on paper tape; 0 word on disk). Writing an ET to disk causes the linker to ignore the remainder of the file.

▶ GENET [G]

Copies the subroutine to which the pointer is currently positioned and follows it with an end-of-tape mark. The pointer moves to the next subroutine. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied, each followed by an end-of-tape mark. When the bottom of the input file is encountered, EDB displays .BOTTOM. on the terminal.

▶ OMITET [G]

Copies the subroutine to which the binary location pointer is currently positioned. The pointer moves to the next subroutine. The optional letter G specifies a global copy; all subroutines from the current position of the pointer are copied. When the bottom of the input file is encountered, EDB displays .BOTTOM. on the terminal.

EDB Error Messages

EDB displays ENTER to show that it is ready to accept commands. Most errors in command input cause EDB to display a question mark (?). Other messages are listed below.

- BAD OBJECT FILE (FRDBIN)

Usually indicates that you have specified a source file, rather than an object (.BIN) file, as the input file. EDB attempts to continue processing by ignoring the remainder of the input file.

- BAD PARAMETERS (EDB)

Indicates an error while locating an input file, an output file, or a replace file; or, indicates an erroneous usage of EDB. EDB terminates.

- ERROR WHILE WRITING

A file system error occurred while EDB was trying to write the contents of an object file. EDB terminates.

EXAMPLES

Creating a Library of Subroutines

The following example creates a library from the files FILE1.BIN, FILE2.BIN, FILE3.BIN, and FILE4.BIN. Each file contains a single module, although FILE1.BIN and FILE2.BIN contain multiple entrypoints. The example shows the EDB commands to list the entrypoints of each file, plus the commands necessary to combine them into a library file, LIBEXP.BIN.

```
OK, EDB FILE1.BIN
[EDB rev 19.4]
ENTER, FIND ALL
ENT1A                              ENT1B
ENT1C
.BOTTOM.
ENTER, NEWINF FILE2.BIN
ENTER, FIND ALL
ENT2D                              ENT2E

.BOTTOM.
ENTER, NEWINF FILE3.BIN
ENTER, FIND ALL
ENT3G
.BOTTOM.
ENTER, NEWINF FILE4.BIN
ENTER, FIND ALL
ENT4H
.BOTTOM.
ENTER, OPEN LIBEXP.BIN
ENTER, NEWINF FILE1.BIN
ENTER, RFL
ENTER, COPY ALL
ENT1A                              ENT1B
ENT1C
.BOTTOM.
ENTER, INSERT FILE2.BIN
ENTER, INSERT FILE3.BIN
ENTER, INSERT FILE4.BIN
ENTER, SFL
ENTER, QUIT
OK,
```

After a library is created, LIBEDB can be run on it to speed its linking time.

Displaying Entrypoints

Notice the difference between the terminal output in VERIFY and TERSE modes. ENT5A and ENT6A are both entrypoints of the module in the file FILE5.BIN; ENT5A is the name of the procedure, ENT6A is the name of an alternate entrypoint to the ENT5A procedure. In TERSE mode, only ENT6A is listed. (The compiler in this case emits the external name for the alternate entrypoint before it emits the external name for the procedure; therefore, ENT6A is listed first.) For example:

```
OK, EDB FILE5.BIN
[EDB rev 19.4]
ENTER, FIND ALL
ENT6A                          ENT5A

.BOTTOM.
ENTER, TOP
ENTER, TERSE
ENTER, FIND ALL
ENT6A
.BOTTOM.
ENTER, QUIT
OK,
```

Replacing an Object Module in the Library

The library file created above, LIBEXP.BIN, is edited to replace the module containing entry point ENT3G with the module in NFILE3.BIN containing entry points ENT3F and ENT3G. The output file is LIBNEW.BIN.

```
OK, EDB NFILE3.BIN
[EDB rev 19.4]
ENTER, FIND ALL
ENT3F                          ENT3G

.BOTTOM.
ENTER, QUIT
OK, EDB LIBEXP.BIN LIBNEW.BIN
[EDB rev 19.4]
ENTER, REPLAC ENT3G NFILE3.BIN
<RFL>                          ENT1A
ENT1B                          ENT1C
ENT2D                          ENT2E
ENT3G
ENTER, COPY ALL
ENT4H                          <SFL>

.BOTTOM.
ENTER, QUIT
```

```
OK, EDB LIBNEW.BIN
[EDB rev 19.4]
ENTER, FIND ALL
<RFL>                          ENT1A
ENT1B                          ENT1C
ENT2D                          ENT2E
ENT3F                          ENT3G
ENT4H                          <SFL>

.BOTTOM.
ENTER, QUIT
OK,
```

Sample Use of LIBEDB

In this example, the file LIBEXP.BIN is processed by LIBEDB, producing a SAM file named FAST_LIBEXP.BIN.

```
OK, RESUME LIB>LIBEDB
[LIBEDB rev 19.0]

SOURCE FILE, DESTINATION FILE, PARAMETER
WHERE: PARAMETER = 0 - DESTINATION FILE SAM
       PARAMETER = 2000 - DESTINATION FILE DAM
$ LIBEXP.BIN,FAST_LIBEXP.BIN,0
OK,
```

APPENDIX

A

Converting Programs That Use Register Settings

Some existing static-mode programs use register settings to select options for the program. Register settings set the initial values of R-mode and V-mode registers for static-mode programs by setting values in the RVEC (Register VECtor) for the user. (See the PRIMOS Commands Reference Guide for more information on RVEC and register settings.)

While using register settings to select program options is obsolete, having been replaced by the more legible and flexible command line options (such as -LISTING, -XREF, and so on), register settings do offer the advantage of being able to change the default options for a program without having to recompile or reload it.

For example, to change the register settings for a program named NRSL, you might type:

```
RESTORE NRSL.SAVE
SAVE NRSL.SAVE 3/14520
```

This command sequence would change the initial value of the A register for NRSL from its original value of 120 to 14520. This might have the effect of enabling more options by default; users subsequently invoking the program would not have to specify those options.

Converting such a program to an EPF might seem difficult at first, because this feature is not directly supplied by BIND and EPFs. However, a feature exists that is easier to use with BIND and EPFs and that may be a suitable replacement. This appendix shows how to use this feature to provide a somewhat compatible interface for setting the initial values of registers.

First, the appendix provides a short discussion of how the present static-mode program uses the initial values of registers. Then, using that model, the appendix describes how to accomplish the same thing using BIND and EPFs.

HOW THE STATIC-MODE PROGRAM WORKS

The key to the use of initial values for registers by a static-mode program is that its first instructions that reference the appropriate registers must not initialize them before using them, because the command processor has already initialized them. Their values are stored in the first nine halfwords of the static-mode runfile containing the program. The first two of these halfwords are the beginning and ending addresses for the program's memory image; the third halfword is the starting location of the program (the initial value of the P register); and the next four halfwords contain the initial values for the A, B, X, and K registers. The remaining two halfwords are undefined and should be 0.

Therefore, the main entrypoint of a static-mode program that utilizes the initial values of one or more registers usually begins with a STA, STL, or STX instruction if written in PMA, or with a call to the GETA or GETL subroutine if written in FTN. (GETA stores the value in the A register into the INTEGER*2 argument passed to it, while GETL stores the value in the L register, which is the A and B registers concatenated, into the INTEGER*4 argument passed to it.)

Then, the main entrypoint uses the values retrieved from the registers as the initial, or default, values for option settings in the program. Typically, the program then reads options from the command line, recording any options it finds there on top of the initial option settings. (Thus, command line options, when specified, override the initial values.)

In addition, the user may use register settings on the command line (such as RESUME NRSL 3/10120) instead of command line options. The use of this obsolete method of specifying program options is guaranteed to confuse and bewilder anybody who tries to understand the command file written by the user to invoke the program. (Such a user rarely builds a CPL program for the purpose.) These register settings, when specified on the command line, override the settings in the RVEC for the static-mode program image, and hence replace the initial values for the registers.

HOW TO ACHIEVE THIS FUNCTIONALITY IN AN EPF

To make the default options for an EPF tailorable on a per-system basis, you build a CPL program that replaces the RESUME/SAVE command sequence shown at the beginning of this appendix; in addition, you convert your static-mode program by changing the way it obtains the initial values of the registers.

The CPL Program

The CPL program performs the following tasks:

1. It determines the default options desired by the user, either by accepting the baroque register settings used for the static-mode version of the program or by reading command line options typed by the user.

2. It compiles a small FTN subroutine called NGETA or NGETL that stores the numeric equivalents to the desired default options into the passed argument, either an INTEGER*2 (NGETA) or an INTEGER*4 (NGETL) argument.

3. It invokes BIND and links the EPF using the subcommand LOAD program.RUN.

4. It uses the RELOAD subcommand to relink the NGETA or NGETL subroutine just compiled into the EPF just linked.

5. It uses the FILE subcommand to write to disk the new version of the EPF with modified default options.

Converting the Static-mode Program

You also convert your static-mode program to obtain the initial values for the registers by calling a subroutine named NGETA or NGETL, depending upon whether the program uses the initial value for the A register or for both the A register and the B register.

Then, you write a subroutine named NGETA or NGETL in FTN that you link with your program. The subroutine sets the passed number to the standard default option settings as numbers and returns to the caller.

The Result

The result you have is a program EPF that obtains its initial register values by calling NGETA or NGETL, an internal subroutine that returns standard values for the registers in the argument provided. The rest of your program operates as it did before.

If someone wishes to tailor your program for their needs, they need only invoke the CPL program you have supplied. It obtains the desired default options from the user, and compiles a new version of NGETA or NGETL that supplies the new initial values instead of the standard values. The CPL program then relinks the newly compiled NGETA or NGETL module into the existing EPF, and now that program EPF uses the new defaults.

A Sample Case

A sample CPL program that performs this conversion, along with the corresponding copy of NGETA and NGETL, follows.

```
&args prog:tree; areg:oct=120

&if [null %prog%] &then &return 1 &message Requires program name.

&if [index %prog% .RUN] ^= [calc [length %prog%] - 3] ~
    &then &s prog := %prog%.RUN
&data ed
      SUBROUTINE NGETA(I)
      INTEGER*2 I
      I=:%areg%
      RETURN
      END

      FILE NGETA.FTN
      &end

ftn ngeta -dynm -dclvar

bind -load %prog% -reload ngeta

delete ngeta.bin
```

The oct=120 in the first line simply sets the default value for the initial A-register setting if the user does not specify it. It should be the same value with which you ship the program EPF.

As you can see from the above sample CPL program, the sample NGETA.FTN module is quite simple:

```
      SUBROUTINE NGETA(I)
      INTEGER*2 I
      I=:value
      RETURN
      END
```

Here, value is the standard initial A-register value. The NGETL.FIN module is as follows:

```
SUBROUTINE NGETL(L)
INTEGER*2 L(2)
L(1)=:value1
L(2)=:value2
RETURN
END
```

Here, value1 and value2 are the standard initial A-register and B-register values, respectively. If your program expects an initial value for the B register, you should use the copy of NGETL shown above and modify the CPL program shown earlier accordingly. (For example, it should take two octal arguments, one for the A register and one for the B register.)

If the Main Entrypoint Is a PMA Program

If the main entrypoint of your program is written in PMA, then you must change the STA or STL instruction at the beginning to a CALL NGETA or CALL NGETL followed by AP INIT_REG_SETTING,SL (where INIT_REG_SETTING was the target of the STA or STL instruction). If the program also expects an initial value for the X register, add a third octal argument to the CPL program and a second argument to NGETL to pass the X register value, and call NGETL with a second argument from the PMA module that stores the value returned in the second argument in the destination of the original STX instruction.

If the PMA program does not start off with STA, STL, or STX instructions, but instead uses instructions that test the registers in various ways (such as SAR, SAS, BEQ, CAS, and so on), simply insert the call to NGETA or NGETL in front of the instructions, then code a LDA, LDL, or LDX instruction to load the registers with the initial values retrieved from NGETA or NGETL.

INDEX

Index

A

Addresses,
  actual, 1-10, 9-2
  ECB in the BIND map, 9-6
  form of, 9-2
  imaginary, 1-10, 1-14, 9-2
  link frame in the BIND map, 9-7
  LIST_EPF command, 9-3
  mapping of, 9-1
  offsets in, 9-2
  procedure code in the BIND map, 9-6
  segment numbers in, 9-2
  stack frame in DUMP_STACK command, 9-9

Arguments to program EPFs, 1-16

B

.BIN file, 3-6, 3-7

Binary editors, 10-1

BIND, 1-2, 1-8
  benefits of using, 1-9
  DYNT subcommand, 5-5

BIND (continued)
  ENTRYNAME subcommand, 3-15
  entrypoint subcommand, 6-8
  initialization of static data, 1-19
  LIBRARY subcommand, 3-11
  linking object files, 3-7
  MAIN subcommand, 3-15, 5-5
  MAP subcommand, 9-5
  RESOLVE_DEFERRED_COMMON subcommand, 3-15
  SYMBOL subcommand, 3-11, 8-2, 8-4
  treatment of common area, 3-11, 3-15
  treatment of IPs, 3-10, 3-11
  use of segment numbers, 3-10

BIND map, 9-5 to 9-7
  determining ECB addresses, 9-6
  determining link frame addresses, 9-7
  determining procedure code addresses, 9-6

Building shared programs with SEG, 1-8

C

Command level breadth, 5-3

Common area, 3-10, 8-1
  defining a shared, 8-2
  initialization of, 3-11
  treatment of by BIND, 3-11, 3-15

Common blocks and dynamic link, 2-4

CP$ subroutine, 3-16

CPL functions and program EPFs, 1-16

D

DATA segment, 3-7, 3-10, 3-19
  access to, 3-16

Deallocation of dynamic memory, 3-32

Deallocation of library EPFs, 3-32

Debugging an EPF,
  BIND command, 1-18
  DBG command, 3-35
  DUMP_STACK command, 1-18
  LIST_EPF command, 1-18
  other useful commands, 1-19
  setting breakpoints, 1-18
  VPSD command, 1-18, 9-6

Debugging information in EPFs, 3-3, 3-7

Displaying common area addresses, 3-15

DUMP_STACK command, 9-9

Dynamic link, 5-5
  common blocks and, 2-4
  definition of, 2-2
  sample session, 2-4
  snapping, 2-3, 3-21

Dynamic link (continued)
  to entry points in PRIMOS, 3-22
  to entrypoints in Application Library, 3-24
  to entrypoints in PRIMOS, 3-26
  to static-mode libraries, 3-28

Dynamic linking mechanism, 1-3, 2-1, 3-6, 3-19
  advantages, 2-1

Dynamic memory, 1-9
  deallocation of, 3-32
  in EPFs, 3-3

DYNT, (See also Dynamic link)
  as a subcommand of BIND, 5-5

E

ECB (entry control block), 1-3
  information contained in, 1-4

EDB binary editor, 10-2 to 10-6
  error messages, 10-5
  obsolete commands, 10-5
  subcommands, 10-3

Entry control block (See ECB)

ENTRY$.SR, 1-3

ENTRYNAME,
  as a subcommand of BIND, 3-15

Entrypoint, 2-2
  as a subcommand of BIND, 6-8
  determining, for library EPFs, 6-5
  invocation, 3-19
  main, of a program EPF, 5-4, 5-5
  modifying the search list of, 6-12, 6-13
  reserved names, 6-5
  subroutine, declaring, 6-8

Entrypoint search list, 6-12, 6-13, 6-32
advanced use of, 6-37
default, 6-32
examining, 6-38

EPF, (See also Library EPF; Process-class library EPF; Program EPF; Program-class library EPF)
benefits of, 1-9
cache, 1-18, 3-34
coding guidelines for, 7-1
copies of link frame, 3-4
debugging information, 3-3
debugging of, 1-18, 3-35
definition of, 1-2
dynamic memory, 3-3
information contained in, 1-18
invocation by CP$ subroutine, 3-16
invocation by EPF$RUN subroutine, 3-16
invocation, forms of, 3-16
library, 1-3
life of an, 3-5 to 3-34
linkage text, 3-2
mapped, 3-16
mechanism, 3-1
multiple invocations of, 3-34
new versions, 1-2, 3-30, 3-34
old versions, 1-2, 3-34
organization of, 3-2
procedure code, 3-2
program, 1-3
reason for, 1-4
removing from memory, 1-16, 3-6, 3-30
restrictions on writing in PMA, 7-10 to 7-16
.RPn suffix, 1-2
.RUN suffix, 1-2
running a remote, 3-36
simultaneous use of, 3-35
stack space, 3-3
(See also Stack frame)
static information and, 4-7
suspending and restarting, 1-17
termination of, 3-6, 3-30, 3-31
types of, 1-3
unmapping, 3-34

EPF (continued)
writing in high-level languages, 7-1
writing in PMA, 7-2

EPF generation and use,
phase 1 (compilation or assembly), 3-7
phase 10 (removal), 3-33
phase 2 (linking), 3-7
phase 3 (invocation), 3-15
phase 4 (mapping), 3-16
phase 5 (linkage allocation), 3-16
phase 6 (linkage initialization), 3-19
phase 7 (entrypoint invocation), 3-19
phase 8 (dynamic links snapped), 3-21
phase 9 (termination), 3-30
phases in, 3-6
stages in, 3-5

EPF$ALLC subroutine, 3-6, 3-16

EPF$DEL subroutine, 3-6, 3-33

EPF$INIT subroutine, 3-6, 3-19

EPF$INVK subroutine, 3-6, 3-19

EPF$MAP subroutine, 3-6, 3-16

EPF$RUN subroutine, 3-5, 3-16

Executable program format (See EPF)

Expanded listings, 9-13

External linkage information, 3-7

F

Faulted IP, 1-3, 2-2, 3-11, 3-19, 3-21, 6-17
how to avoid sharing, 4-10
sharing of, 4-9

Freeing segments of R-mode programs, 1-7

I

I-mode programs, 1-5

Imaginary addresses and EPF sharing, 1-14

Impure code, 1-13
  separation of pure code from, 1-12, 7-2

IMPURE segment, 3-7, 3-10, 3-19
  access to, 3-16

Indirect pointer (See IP)

Initialization,
  of variables, 1-19
  shared data, 8-3, 8-4

Invoking an EPF, 1-3, 3-15, 5-1, 5-2
  subroutines for, 3-5

IP (indirect pointer), 1-3, 3-7
  faulted, 1-3, 2-2, 3-11, 3-21, 6-17
  how to avoid sharing faulted, 4-10
  resolution of at runtime, 3-10
  sharing of faulted, 4-9
  treatment of by BIND, 3-10, 3-11

L

LB (See Linkage base)

LIBEDB binary editor, 10-1

LIBRARY,
  as a subcommand of BIND, 3-11
  external references resolved by, 3-11

Library EPF, 1-3, 1-9
  assembling the PMA entrypoint file for, 6-10
  building a PMA entrypoint file for, 6-8, 6-9
  choosing the right type of, 6-4, 6-14, 6-15
  coding a subroutine for, 6-4
  compiling a subroutine for, 6-4
  deallocation of, 3-32
  definition of, 6-2
  determining class requirements of, 6-29
  determining entrypoints of, 6-4
  installing a library file, 6-11
  installing the library EPF, 6-11
  invoking, 1-3
  linking subroutines of, 6-7
  modifying the entrypoint search list, 6-12
  process-class, 3-32, 3-33
  program's view of, 6-4
  program-class, 3-32, 3-33
  programmer's view of, 6-2
  restriction on class mixing of, 6-16
  restriction on use of language I/O, 6-17
  steps in building, 6-2, 6-4 to 6-13
  storage allocation issues, 6-41
  storing data in linkage area of, 6-17
  using DBG on, 6-30, 6-31
  using EDB to generate a library file, 6-10

Library EPF mechanism, 6-39

Limits on calling program EPFs, 5-3

Link frame, 3-4, 3-5, 3-10

Linkage,
  area, 3-32
  area, storing data in, 6-17, 6-18
  base, 3-4

Linkage (continued)
fault, 2-4
initialization, 3-19, 6-18
text, 3-7
text, in EPFs, 3-2
text, in subroutines, 3-4

Linking,
loaders, history of, 1-4
purpose of, 3-7
utilities, 1-2

LIST_EPF command, 9-3

LIST_SEGMENT command, 9-5

LOAD, 1-2, 1-4

M

MAIN,
as a subcommand of BIND, 3-15, 5-5

MAP,
as a subcommand of BIND, 9-5

Mapping an EPF, 3-16

Maps and addresses, 9-1

Memory,
allocation of, 1-10, 1-11
dynamic, 1-9
static, 1-9

Multiple invocations of an EPF, 3-34

O

Object file, 3-7

P

PB (See Procedure base)

PCL instruction, 3-4, 3-7, 3-19

PMA,
restrictions for EPF execution, 7-10
writing EPFs in, 7-2 to 7-10

PROC segment, 3-7, 3-10
access to, 3-15

Procedure,
base, 3-4
code in EPFs, 3-2
code in subroutines, 3-4
frame, 3-4
main, of a program EPF, 5-4
management, 3-4
text, 3-7

Process-class library EPF, 3-32, 3-33, 6-41
choice of, 6-14
link sequence for, 6-7
restrictions on use of, 6-14
using for shared data, 8-5

Program,
I-mode, 1-5
R-mode, 1-4, 1-5
S-mode, 1-5
static-mode, 1-4
V-mode, 1-5

Program EPF, 1-3, 1-9
arguments to, 1-16, 5-4
command line preprocessing, 1-17
CPL functions, 1-16
data returned from, 5-3
data supplied to, 5-3
definition of, 5-1
invoking, 1-3, 5-1, 5-2
invoking program's view of, 5-2
limits on calling, 5-3
main entrypoint of, 5-4, 7-2
main procedure of, 5-4
programmer's view of, 5-1
stacking of, 1-17
user's view of, 5-2
writing the main program, 5-4

Program-class library EPF, 3-32, 3-33, 6-40
choice of, 6-14
link sequence for, 6-7

PRTN instruction, 3-4

Pure code,
  separation of impure code from, 1-12, 7-2
  sharing of, 1-13

R

R mode, 1-4

R-mode programs, 1-4, 1-5
  freeing segments of, 1-7

Removal of EPFs, 1-16, 3-30
  process-class library, 3-33
  program EPF, 3-33
  program-class library, 3-33

Replacing static-mode libraries, 3-30

Reserved entrypoint name list, 6-6

Reserved entrypoint names, 6-5
  list of, 6-6

RESOLVE_DEFERRED COMMON,
  as a subcommand of BIND, 3-15
  to display common area address, 3-15

.RPn suffix, 1-2

.RUN suffix, 1-2

Running a remote EPF, 3-36

S

S mode, 1-5

SB (See Stack base)

Search rule, 1-3, 6-12, 6-33 to 6-36

SEG, 1-2, 1-4
  building shared programs, 1-8
  for invoking V- or I-mode programs, 1-7
  for shared procedure segments, 1-8
  generating static-mode images, 1-8

Segment access,
  to DATA segments, 3-16
  to IMPURE segments, 3-16
  to PROC segments, 3-15

Segment number,
  for IMPURE and DATA segment, 9-3
  for PURE segment, 9-3
  in addresses, 9-2
  sign of, 9-2, 9-3
  use of by BIND, 3-10

Segments,
  shared system-wide, 1-8, 8-3
  static, 8-3

Separation of pure and impure code, 1-13, 1-15

Shared applications, (See also Shared programs)
  effect of EPFs on existing, 4-8

Shared data, 8-1 to 8-7
  determining the address of, 8-2
  how to update atomically, 8-7, 8-8
  initializing, 8-3, 8-4
  PMA subroutines for updating, 8-9 to 8-11
  process-wide, 8-1
  system-wide, 8-1
  using a process-class Library EPF for, 8-5

Shared programs,
  deleting old versions, 1-16
  installing new versions, 1-8, 1-16
  using SEG to build, 1-8

Shared system-wide segments, 1-8

Sharing faulted IPs, 4-9
how to avoid, 4-10

Sharing of pure code, 1-13

Simultaneous use of an EPF, 3-35

Snapping dynamic links, 2-3, 3-21

Stack base, 3-4

Stack frame, 3-4
addresses of in DUMP_STACK command, 9-9
locating procedure, 9-10

Stack header, 3-4

Stack space in EPFs, 3-3

Stack space in subroutines, 3-4

Stacking program EPFs, 1-17

Static data, 3-7

Static information and EPFs,
command line information, 4-7
error information, 4-7

Static memory, 1-9

Static-mode applications, (See also Static-mode program)
conversion strategy, 4-1
relation of EPFs to, 4-1
restriction on EPF use of, 4-2
suspending and continuing, 4-2

Static-mode library, 3-28
dynamic link to, 3-28
replacing, 3-30
restriction on EPF use of, 4-4

Static-mode program, 1-5
(See also Static-mode applications)
characteristics of, 1-5 to 1-7

Subroutine, (See also library EPF)
converting nonreentrant to reentrant, 6-21 to 6-25

Subroutine (continued)
determining class requirements of, 6-15, 6-16
determining the use of static data by, 6-17, 6-18
linkage text, 3-4
nonreentrant process-class, 6-20
optimizing conversion approach to, 6-25 to 6-28
organization of, 3-4
procedure code, 3-4
process-class, 6-15
program-class, 6-15
stack space, 3-4
storing data in linkage area of, 6-18

Subroutine libraries, 2-1
types of, 2-2

Subroutine not found condition, 2-4

Subroutines,
dynamic linking of, 2-1

Subroutines for invoking EPFs, 3-5

SYMBOL,
as a subcommand of BIND, 3-11, 8-2, 8-4
to locate common areas, 3-11

T

Terminating an EPF, 3-6, 3-30, 3-31

Types of EPFs, 1-3

U

Unmapping an EPF, 3-34

V

V-mode programs, 1-5

VPSD command, 1-18, 9-8

SURVEY

READER RESPONSE FORM

DOC10055-1LA Advanced Programmer's Guide, Volume I First Edition

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: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

READER RESPONSE FORM

DOC10055-1LA Advanced Programmer's Guide, Volume I First Edition

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: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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

READER RESPONSE FORM

DOC10055-1LA Advanced Programmer's Guide, Volume I First Edition

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: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

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