Prime

A Prime Company

Advanced Programmer's Guide I: BIND and EPFs

Revision T3.0–23.0

DOC10055–2LA

Advanced Programmer's Guide I: BIND and EPFs

Second Edition

Glenn Morrow

This manual documents the software operation of the PRIMOS operating system on 50 Series computers and their supporting systems and utilities as implemented at Master Disk Revision Level 23.0 (Rev. 23.0) and Translator Family Revision Level T3.0.

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

Printing History

Preliminary Edition (DOC9229-1LA) January 1985 for Revision 19.4.0
First Edition (DOC10055-1LA) November 1985 for Revision 19.4.2
Second Edition (DOC10055-2LA) September 1990 for Revision T3.0–23.0

Credits

Editorial: Mary Skousgaard
Production Support: Sonya Zegarra
Technical Support: Julie Cyphers, Rosemary Crowley
Illustration: Carol Smith
Production: Judy Gordon

How to Order Technical Documents

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

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

  1-800-343-2533

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

- International customers contact your local Prime subsidiary or distributor.

PRIME SERVICE℠

To obtain service for Prime systems

- United States customers call toll free at

  1-800-800-PRIME

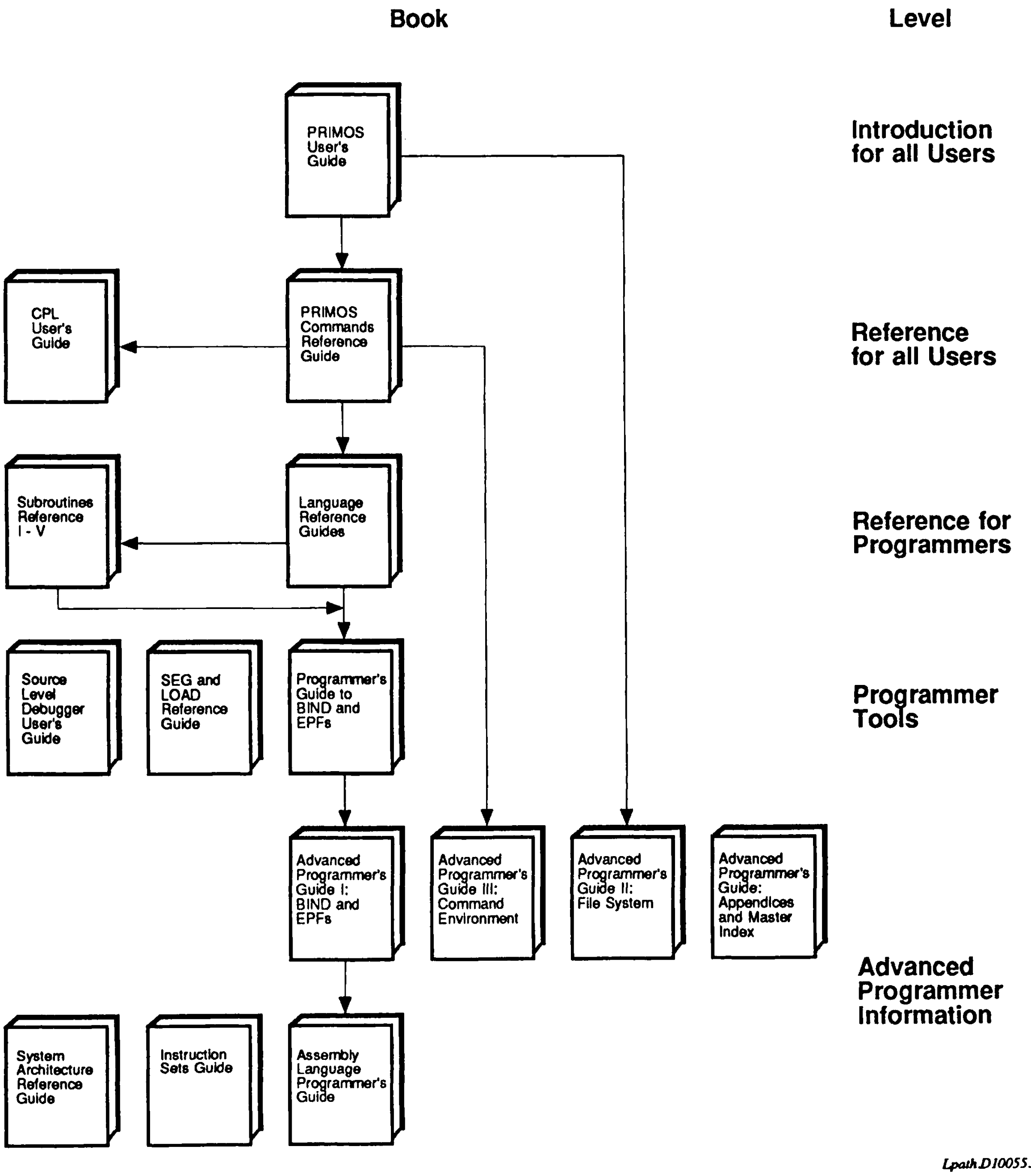
- International customers contact your Prime representative.

Surveys and Correspondence

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

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

Reading Path for PRIMOS Documentation


Book
Level
PRIMOS User's Guide
Introduction for all Users
CPL User's Guide
PRIMOS Commands Reference Guide
Reference for all Users
Subroutines Reference I - V
Language Reference Guides
Reference for Programmers
Source Level Debugger User's Guide
SEG and LOAD Reference Guide
Programmer's Guide to BIND and EPFs
Programmer Tools
Advanced Programmer's Guide I: BIND and EPFs
Advanced Programmer's Guide III: Command Environment
Advanced Programmer's Guide II: File System
Advanced Programmer's Guide: Appendices and Master Index
Advanced Programmer Information
System Architecture Reference Guide
Instruction Sets Guide
Assembly Language Programmer's Guide
LpathD10055.2LA

Contents

About This Book

The *Advanced Programmer's Guide* is a four-volume series that provides technically sophisticated information for systems-level programmers. This series supplements basic reference information found in other PRIMOS® manuals.

The books in this series are intended for programmers who are experienced with the PRIMOS operating system and 50 Series™ systems. In addition, you should be experienced in at least one high-level programming language supplied by Prime (preferably PL/I, C, or FORTRAN-77).

The *Advanced Programmer's Guide* series consists of four volumes:

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

The four volumes of the *Advanced Programmer's Guide* can be ordered as a set using DCP10171.

Specifics of This Volume

This volume describes Prime's Executable Program Format (EPF), the standard format for executable programs and subroutine libraries for all languages supported by Prime®. This manual assumes a working knowledge of the BIND linker, which is used to build EPFs. The BIND linker is described in the *Programmer's Guide to BIND and EPFs.*

- Chapter 1 describes the four types of EPFs: dynamic program EPFs, dynamic library EPFs, registered program EPFs, and registered library EPFs. It also outlines the basic steps used to create an EPF.

- Chapters 2 and 3 provide background information about the essential principles of EPFs: dynamic memory allocation and dynamic linking to called routines.
- Chapters 4, 5, and 6 provide detailed instructions for creating and using the different types of EPFs.
- Chapters 7 and 8 provide in-depth information useful for analyzing and debugging EPFs in memory.
- Chapter 9 documents the EDIT_BINARY binary file editor.
- Appendix A describes special considerations for writing EPFs in PMA, Prime's assembly language.
- Appendices B and C provide reference information on obsolete but still supported facilities: the LIBEDB and EDB binary file editors and static-mode programs.
- Appendix D lists the runtime libraries that Prime supplies as registered library EPFs.

Specifics of the Series

The *Advanced Programmer's Guide* series divides information among the volumes of the set as follows:

- Volume I: BIND and EPFs (this volume) describes Executable Program Formats (EPFs), including registered EPFs, and describes the EDIT_BINARY binary file editor.
- Volume II: File System describes the PRIMOS file system. It provides detailed information about the file server, access rights, search rules, and data and text manipulation in file system objects.
- Volume III: Command Environment describes how to use EPF initialization routines and how to invoke a user program as a command, subroutine, or function from a user program or from PRIMOS command level.
- Appendices and Master Index provides appendix material applicable to all of the volumes in this document set. It lists the standard error codes used by PRIMOS, along with their messages and meanings. It describes the new features of recent PRIMOS revisions that may be of interest to advanced programmers. Finally, it provides a Master Index to all four volumes of the *Advanced Programmer's Guide* series.

This series describes the lowest-level interfaces supported by PRIMOS and its utilities. It is designed for systems-level programmers 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 series for best results. Most of the information in this series 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.

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

The above interfaces are described in other manuals, such as language reference manuals and the *Subroutines Reference* series.

References

The basic document set for the PRIMOS operating system is shown in the Reading Path for PRIMOS Documentation on page iv of each manual. This illustration shows the manuals and their intended audience. Lines between manuals show the order in which these manuals are commonly used to locate increasingly detailed information about a topic.

Users of this series should be familiar with the *PRIMOS User's Guide* (DOC4130-5LA), which contains information on system use, directory structure, the condition mechanism, CPL files, ACLs, global variables, and how to load and execute files with external subroutines. New information for Rev. 23.0 can be found in the *PRIMOS User's Release Document* (DOC10316-1PA) and the *Rev. 23.0 Software Release Document* (DOC10001-7PA).

You should use the *Advanced Programmer's Guide* along with the standard PRIMOS references: the *PRIMOS Commands Reference Guide* (DOC3108-7LA updated by RLN3108-71A) and the five-volume *Subroutines Reference* series:

- *Subroutines Reference I: Using Subroutines* (DOC10080-2LA updated by UPD10080-21A)
- *Subroutines Reference II: File System* (DOC10081-2LA)
- *Subroutines Reference III: Operating System* (DOC10082-2LA)
- *Subroutines Reference IV: Libraries and I/O* (DOC10083-2LA)

- *Subroutines Reference V: Event Synchronization* (DOC10213-1LA updated by UPD10213-11A)

Users of this series should be familiar with Prime system architecture, as described in the *50 Series Technical Summary* (DOC6904-2LA) and the *System Architecture Reference Guide* (DOC9473-2LA).

Users of this volume should also be familiar with the following Prime publications:

- *Programmer's Guide to BIND and EPFs* (DOC8691-1LA) and its updates UPD8691-11A and UPD8691-12A
- *SEG and LOAD Reference Guide* (DOC3524-192L)

System Administrators installing registered EPF libraries supplied by Prime should consult the *Rev 23.0 Software Installation Guide* (IDR10176-3XA).

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

Prime Documentation Conventions

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

Convention	*Explanation*	*Example*
Uppercase	In command formats, words in uppercase bold indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	**LIST_EPF**
Italic	Variables in command formats, text, or messages are indicated by lowercase italic.	**FILE** *my_prog*
Abbreviations in format statements	If a command or option has an abbreviation, the abbreviation is placed immediately below the full form.	**DEFAULT_DYNT_TYPE** **DDT**
Brackets	Brackets enclose a list of one or more optional items. Choose none, one, or several of these items.	$\textbf{LD} \begin{bmatrix} \textbf{-BRIEF} \\ \textbf{-SIZE} \end{bmatrix}$
Braces	Braces enclose a list of items. Choose one and only one of these items.	$\textbf{CLOSE} \begin{Bmatrix} \textit{filename} \\ \textbf{-ALL} \end{Bmatrix}$

Convention	*Explanation*	*Example*
Braces within brackets	Braces within brackets enclose a list of items. Choose either none or only one of these items; do not choose more than one.	**BIND** $\left[\begin{Bmatrix} \textit{pathname} \\ \textit{options} \end{Bmatrix}\right]$
Monospace	Identifies screen output, user input, prompts, and messages.	`address Connected`
User input in examples	In examples, user input is underscored but system prompts and output are not.	`OK, RESUME MY_PROG`
Hyphen	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	**DYNT -SHARED**
Ellipsis	An ellipsis indicates that you have the option of entering several items of the same kind on the command line.	*pdev-1* [...*pdev-n*]
Subscript	A subscript after a number indicates that the number is not in base 10. For example, the subscript 8 is used for octal numbers.	200_8
Parentheses	In command or statement formats, you must enter parentheses exactly as shown.	**DIM** *array* (*row, col*)

Introduction to EPFs

1

An **EPF (Executable Program Format)** is the standard format for executable programs and subroutine libraries for all languages supported by Prime. Compiling (or assembling) your program produces a binary object code file. You link this binary file using BIND, a dynamic linker. The output file created by BIND is an EPF. An EPF is an executable code file.

EPFs are dynamic. EPF memory allocation is handled dynamically during program execution. That is, PRIMOS establishes addressing to whatever locations in memory are available, rather than requiring pre-specified locations in memory.

Most programs contain calls to external routines. These external routines are stored in separately compiled binary libraries (.BIN files). When you use BIND to build an EPF, it establishes links from your program to these external routines. For example, if your program calls an external subroutine, BIND establishes a link between the call statement and the subroutine. These links are dynamic. A **dynamic link** contains information that enables PRIMOS to locate the subroutine when needed; it does not contain the actual address of the subroutine.

PRIMOS also supports separate tools (SEG and LOAD) that create executable programs that contain **static links**. Static linking should not be used for new program development. Appendix C of this guide describes how to convert existing static programs to EPFs. Static linking is not described in this guide; refer to the *SEG and LOAD Reference Guide* for further details.

For basic information on programming with EPFs, see the *PRIMOS User's Guide*. For information on using the BIND linker, see the *Programmer's Guide to BIND and EPFs*.

EPFs simplify the creation, installation, and maintenance of user-written programs. PRIMOS takes care of loading, sharing and most memory management so you can concentrate on program functionality. You can create a variety of EPF types to match the specific requirements of your application.

Figure 1–1 shows the relationships among the four types of EPFs: dynamic program EPFs, dynamic library EPFs, registered program EPFs, and registered library EPFs. These EPF types are described in the sections of this chapter that follow.

Dynamic Program EPF	Dynamic Library EPF
Registered Program EPF	Registered Library EPF

Figure 1–1. The Four EPF Types

Information presented in Chapters 4, 5, and 6 explains the shared characteristics of the four EPF types and builds on the relationships shown in Figure 1–1.

Dynamic and Registered EPFs

There are two main types of EPFs: dynamic EPFs and registered EPFs. Both dynamic EPFs and registered EPFs are created using the BIND linker and both types perform dynamic memory allocation and contain dynamic links.

Dynamic EPFs

A dynamic EPF resolves its dynamic links during program execution. Prior to execution, a dynamic EPF is stored in the file system. When a user invokes a dynamic EPF, PRIMOS automatically maps the EPF into special dynamic segments set aside in that user's private address space. Then, during execution, PRIMOS resolves the dynamic links to other resources called by the dynamic EPF.

To create a dynamic EPF, you use the BIND linker. The BIND linker creates dynamic EPFs by default.

Registered EPFs

A registered EPF resolves some of its dynamic links prior to execution, and resolves other dynamic links during program execution. Registered EPFs are a shared resource for all users on the system. They are maintained in shared address space and stored in a special registered EPF database. As of PRIMOS Revision 23.0 and Translator Family Revision T3.0, most system and language libraries supplied by Prime are registered EPFs (a complete list of these libraries is found in Appendix D). As of the Translator Family Rev. T3.0, BIND supports new options that permit you to create your own registered EPFs.

Creating a registered EPF is a two-step process: first you create the EPF using the BIND linker, specifying that the EPF will be a registered EPF. Then the System Administrator registers the EPF.

Note This book uses the general term **registered EPF** for this type of EPF whether or not the EPF has, in fact, been registered. The term **registrable EPF** is only used when it is essential to refer to the file system object version of the EPF.

When an EPF is registered, PRIMOS automatically allocates space for it from available shared dynamic segments, resolves dynamic links to external routines, and performs a variety of initialization tasks. PRIMOS resolves the remaining dynamic links during execution of the EPF.

A registered EPF remains registered (and occupies system resources) either until the System Administrator unregisters the EPF or until system cold start. A cold start unregisters all registered EPFs.

Registered EPFs are especially useful for programs and libraries that are widely used on a system. Because much of the initialization and resolution of dynamic links is performed at registration time, rather than each time the program is executed, registered EPFs can be more efficient than dynamic EPFs for many applications. Because at least part of a registered EPF is mapped to shared memory, registered EPFs occupy less space in the user's private address space. Chapter 6 discusses registered EPFs and includes information you can use to decide whether a given EPF is a good candidate for registration.

Program and Library EPFs

Both dynamic EPFs and registered EPFs can contain any type of executable code. An EPF can contain an application program, a command, a command function, or a library of subroutines. You create programs, commands, and functions with **program EPFs**. You create subroutine libraries with **library EPFs**.

Program EPFs

Program EPFs are programs with a single main entrypoint. You can invoke a program EPF directly from the command line. You can also invoke a program EPF indirectly from a running program by calling one of the PRIMOS subroutines that invokes a program EPF. Chapter 4 gives specific information about program EPFs.

Library EPFs

Library EPFs are collections of routines. Each library EPF contains one or more entrypoints. Programs invoke the routines in library EPFs by calling the entrypoints. PRIMOS links programs to routines in library EPFs dynamically. Chapter 5 discusses the library mechanism and library EPFs.

BIND establishes dynamic links from an EPF to called routines. The routines called by the EPF reside in runtime libraries. A **runtime library** is a collection of executable routines that can be called by many programs. A runtime library can be a library EPF, a shared static-mode library, or a PRIMOS direct entry.

Routines in a runtime library never become a part of the runfile of the EPF that calls them. Instead, each EPF contains dynamic links that enable PRIMOS to find and execute the routines at runtime. PRIMOS uses your ENTRY$ search list to locate your runtime libraries. Chapter 5, Library EPFs, discusses runtime libraries in detail.

Routines in a library EPF can call other routines in the same library EPF or another library EPF. Because the links between an EPF (of any type) and its called routines are dynamic, you can create program EPFs and library EPFs in any sequence. (Some restrictions apply to the registration of registered EPFs, as described in Chapter 2.)

Library routines can be altered, rebuilt, and relocated on the system without requiring you to relink applications that call them. When updating a library routine, just make sure that you have not altered the parameter interface between the routine and its calling programs.

Creating and Using EPFs

The steps required to create and use an EPF are

1. Create the source code in a high-level language or PMA.
2. Compile or assemble the source code using a Prime compiler or the PMA assembler.
3. Use the BIND linker to create the EPF.
4. Install the EPF in an appropriate location.
5. For registered EPFs, have the System Administrator register the EPF.
6. Execute the EPF.

Source Code

A program EPF contains a main routine and may optionally contain internal subroutines. A library EPF contains only subroutines. Both program EPFs and library EPFs can contain calls to external routines.

For high-level languages there are no restrictions on coding. You can create all types of EPFs with any Prime language. However, not all compilers create code that takes full advantage of registration. See Chapter 6 for information on which languages fully support registered EPFs. Appendix A gives complete information on writing EPFs in PMA.

Compiling

Compile or assemble your source code using a Prime compiler or the PMA assembler. You can create either V-mode (the default) or I-mode object code. If you are creating a registered EPF, be sure to use a version of the compiler that supports registered EPFs.

Linking With BIND

You generate all types of EPFs using the BIND linker. Because of the flexibility of the EPF format, most linking with BIND is straightforward. You use BIND subcommands to specify whether to generate a dynamic or registered EPF and whether it is to be a program or library EPF. BIND generates a dynamic program EPF by default. If you specify that the EPF is to be a registered EPF, the executable file created by BIND is referred to as a **registrable EPF**. A registrable EPF must be registered before it can be executed as a shared program or library. However, you can test a registrable EPF prior to registering it, as described in Chapter 6.

BIND links but does not load your EPF. Instead, BIND organizes your runfile in EPF format so that PRIMOS can load it dynamically at invocation or registration time. This means that you almost never need to concern yourself with the location of the EPF code and data in memory. BIND and PRIMOS automatically take care of allocating memory, loading, and sharing, as well as replacing earlier versions.

The input to BIND is one or more **binary files** (.BIN files). These binary files can be output from a compiler and contain object code for a program or a group of library routines. BIND also takes as input **binary libraries**. A binary library is a binary file created using EDIT_BINARY that contains linkage information from one or more library EPFs. You can set your BINARY$ search rules to enable BIND to locate binary files and binary libraries by filename, rather than supplying a complete pathname. The output from BIND is an EPF, an executable code file (.RUN file). It can be a program EPF or a library EPF.

This guide includes specific information on linking library EPFs in Chapter 5 and registered EPFs in Chapter 6. For a complete reference of all BIND subcommands, refer to the *Programmer's Guide to BIND and EPFs.*

Installing EPFs

Dynamic program EPFs are ready to execute as soon as they have been linked with BIND. PRIMOS automatically takes care of loading when you invoke the EPF. For dynamic program EPFs, the only installation needed is to copy the EPF to a useful location in the file system. If you wish, you can also alter users' COMMAND$ search lists so that the program EPF can be run as a command. For more information on installing dynamic program EPFs, see Chapter 4.

For dynamic library EPFs, install the program in the file system and update either the system or individual users' ENTRY$ search lists. For library EPFs you may also want to create a matching binary library with dynts (dynamic links) to the routines in your library EPF.

Registered EPFs are installed by the System Administrator.

Registering EPFs

A registrable EPF must be registered by the System Administrator. Registration is simple, because PRIMOS automatically takes care of allocating shared memory to registered EPFs. Once an EPF is registered, it is available to all users as a command. For information on registering EPFs, see Chapter 6.

Executing EPFs

Execution of an EPF consists of two steps: invocation and execution. When an EPF is **invoked**, PRIMOS allocates resources and initializes values. As an EPF **executes**, PRIMOS encounters calls to routines, and, if necessary, resolves the dynamic links to those routines.

Users can use the RESUME command to execute any program EPF to which they have sufficient access rights. You can use COMMAND$ search rules to make a dynamic program EPF available as a command. All registered program EPFs are automatically available as commands when using the default COMMAND$ search rules. A program EPF can also be executed using a PRIMOS subroutine call, for example, EPF$RUN.

Library EPFs are executed indirectly. A program EPF calls specific entrypoints in the library EPF.

EPFs and Static-mode Applications

Virtually all static-mode applications can be converted to EPFs. The EPF versions give equal or better performance, allow applications to take advantage of the far more flexible EPF environment, and greatly simplify installation and maintenance. V-mode and I-mode object files can usually be converted simply by relinking with the BIND linker. R-mode object files must be recompiled as V-mode or I-mode. Static-mode programs that share dynamic links to reduce dynamic linking overhead can be converted to registered EPFs with no loss of performance. Appendix C compares the BIND and SEG linkers and discusses converting static-mode programs to EPFs.

2 EPF Principles

This chapter introduces some of the basic elements of EPF technology:

- Dynamic linking
- EPF organization
- EPF mapping

These elements are used by EPFs of all types. You should find the definitions and descriptions given in this chapter useful as you read the discussions of specific applications throughout the rest of this book.

Dynamic Linking

This section describes the logical connections between an EPF and the external routines that it calls.

Dynamic linking creates flexible connections between EPFs and routines in runtime libraries. Dynamic linking keeps EPFs and the routines that they call functionally and physically separate. This greatly simplifies program maintenance. Both program EPFs and library EPFs can contain dynamic links to their called external routines. They do not use dynamic links to their own internal routines.

A dynamic link established by BIND is known as a **dynt**. A dynt specifies the name of a called routine, but does not specify its location. To determine the actual location of a routine, PRIMOS **resolves** the dynamic link. This is also referred to as **snapping the dynt**. PRIMOS snaps these dynts to point to the actual routines either when the EPF is executed or when it is registered.

There are two types of dynts, **per-user dynts** and **shared dynts**. These dynt types are snapped at different times. Both dynamic EPFs and registered EPFs can contain per-user dynts. PRIMOS snaps per-user dynts as they are encountered during program execution. Only registered EPFs can contain shared dynts. PRIMOS snaps shared dynts at registration time, thus reducing dynamic linking overhead during program execution.

This dynamic linking mechanism simplifies EPF development and maintenance and reduces the use of system resources.

Creating Dynamic Links

To execute a called routine, PRIMOS must determine the address of the routine in a runtime library. When the BIND linker builds an EPF, it establishes **dynamic links** to all called routines; it does not install the actual addresses of the called routines in the EPF. Instead, PRIMOS resolves these dynamic links to the actual addresses of called routines as needed.

A dynamic link (or dynt) is a two-part object. It consists of an **Indirect Pointer (IP)** and a character varying string that contains the name of the called routine. PRIMOS modifies the IP when it resolves the dynamic link; therefore BIND stores the IP in a modifiable data segment. PRIMOS does not modify the name string; therefore BIND stores this part of the dynt in a nonmodifiable procedure segment.

The first bit of the IP is the fault bit. BIND automatically sets this fault bit to 1 to tell PRIMOS that the IP does not contain the actual address of the routine. Such an IP is called a **faulted IP**. The rest of the IP contains the address of the name string for the routine.

Resolving Dynamic Links

PRIMOS resolves a dynamic link by resetting the faulted IP with the address of the routine. PRIMOS determines the actual location of the routine by searching runtime libraries, either during program execution (per-user dynts) or during registration (shared dynts).

PRIMOS accesses the user's ENTRY$ search list to determine which runtime libraries to search. PRIMOS searches these runtime libraries in the sequence specified in ENTRY$. Each runtime library contains an **entrypoint table**, which lists its available entrypoints. PRIMOS reads the entrypoint table in each runtime library. When PRIMOS locates an entrypoint name that matches the dynt's name string, PRIMOS snaps the dynt.

Each element in an entrypoint table contains a pointer to an **Entry Control Block (ECB)**. PRIMOS follows this pointer to the entrypoint's ECB. The ECB contains a pointer to the entrypoint itself, the first instruction of the routine. (A **shortcalled routine** stores the address of the entrypoint itself in the entrypoint table; it does not use an ECB.)

When PRIMOS locates the entrypoint, it returns to the dynt in the calling program. PRIMOS replaces the address in the faulted IP with the actual address of the entrypoint, then resets the fault bit to zero. This process is called **snapping the dynt**.

PRIMOS then retries the call by reexecuting the PCL instruction, using the newly modified IP.

Per-user dynts: With dynamic EPFs and the non-shared portions of registered EPFs, PRIMOS snaps dynts as it encounters them during execution. These are per-user dynts. The first time PRIMOS encounters a call to a particular routine during program execution, it checks the indirect pointer. Because the fault bit of the indirect pointer is set, a pointer fault occurs. PRIMOS then attempts to snap the dynt and retry the call.

Shared dynts: With registered EPFs, PRIMOS snaps all shared dynts at registration time. Since these dynts are snapped before execution time, they do not change during program execution and can be shared along with the pure procedure code. Because shared dynts are presnapped, the same program may execute faster as a registered EPF than as a dynamic EPF.

A registered EPF cannot share dynts to routines in a dynamic library EPF. A registered EPF can only share dynts to registered library EPFs and PRIMOS direct entries. At registration time, PRIMOS searches only those library EPFs that have already been registered. At that time PRIMOS snaps the shared dynts to the routines found in those runtime libraries. PRIMOS snaps the per-user dynts in the registered EPF when a user executes the EPF.

Dynamic Linking Errors

When PRIMOS attempts to snap a dynt it may not find the named routine. During program execution, this usually happens because the library containing the routine is not named in the user's ENTRY$ search rules. If PRIMOS cannot successfully snap a dynt at runtime, it signals a LINKAGE_FAULT$ condition. Unless intercepted by a program, this condition results in a display like 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 name of the routine that could not be found, `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!
```

When the System Administrator registers an EPF, PRIMOS attempts to snap all shared dynts. If PRIMOS cannot snap a shared dynt at registration time, it places the EPF in a suspended state. This EPF is not fully registered. PRIMOS does not permit the execution of a registered EPF until all shared dynts have been successfully snapped. If the routine referenced by a shared dynt is in a library EPF that has not yet been registered, PRIMOS cannot snap the shared dynt. When the System Administrator registers this library EPF, PRIMOS automatically snaps the shared dynt in the suspended EPF and changes the state of the suspended EPF to ready. This process is discussed in detail in Chapter 6, Registered EPFs.

EPF Organization

This section describes the physical organization of the different components of an EPF.

Segment Types

BIND organizes EPFs into segments. Which segment holds a given part of an EPF depends mainly on whether the code in that part is modified at runtime. The precise organization varies according to the compiler used, the linking procedure, and the type of EPF. For dynamic EPFs, BIND gathers pure procedure code into one or more shareable **pure procedure segments** and puts linkage and static data into one or more per-user **linkage/data segments**. For registered EPFs, BIND puts shareable linkage and pure procedure code together in shared linakge/data segments and per-user linkage and static data in non-shared linkage/data segments. For both types, BIND places any impure procedure code in non-shared **impure procedure segments**.

When PRIMOS maps an EPF to memory, it allocates space in an appropriate memory area for each segment. Because BIND organizes an EPF into segments, but PRIMOS does not allocate memory space to these segments until later, all EPF segments are known as **dynamic segments**. The EPF Mapping section below explains the process of mapping EPF segments to memory.

EPF Format

The EPF runfile created by BIND is not simply a memory image. EPFs are stored in a generalized format that allows PRIMOS to map them efficiently into any available dynamic segments. This format includes compressed descriptions of linkage/data segments and imaginary addresses.

Linkage/Data Description: BIND stores EPF linkage/data segments as compressed descriptions rather than memory images. BIND classifies linkage/data into a variety of types such as ECBs, IPs, repeated data, and the like. The linkage/data description consists of entries naming specific types along with blocks of actual data for each entry. When PRIMOS maps an EPF, it expands these descriptions to create the contents of each linkage/data segment. To expand a description, PRIMOS fills in templates for each linkage/data type using the blocks of data contained in the description.

- A dynamic EPF runfile contains a memory image of the pure procedure code portion of the EPF and a compressed description of the per-user linkage and data. PRIMOS creates a copy of the per-user portion by expanding this compressed description.
- A registrable EPF runfile contains descriptions of shareable code, linkage, and data. PRIMOS generates the contents of both the shared and per-user segments from these descriptions.

Imaginary Addressing: BIND does not use actual absolute memory addresses when creating an EPF runfile. Instead, it assigns each segment an imaginary segment number and creates **imaginary addresses** consisting of the imaginary segment number and an offset. For example, in a dynamic EPF, the first procedure segment is imaginary segment number +0. The imaginary address of the 1000th address in the first procedure segment is therefore +0/1000. (Note that all addresses are in octal.)

BIND must use imaginary addresses because PRIMOS can map an EPF to any available locations in a user's virtual address space. For example, an IP in per-user linkage that points to a location in the procedure code must use the imaginary address of that location. When PRIMOS maps the EPF and creates a copy of the per-user linkage for a user, it translates the imaginary address to an actual address in the user's address space. If, for example, PRIMOS maps segment +0 in the above example to a user's segment 4771, then an IP that points to the imaginary address +0/1040 is set to point to the actual address 4771/1040.

Note Not all imaginary segments are loaded so that they begin at offset 0 of an actual segment. Therefore, PRIMOS may also adjust the imaginary address offset when it creates the actual address. For example, linkage/data segments can be loaded anywhere in a read/write segment; they could begin at some non-zero offset in an actual segment.

Chapter 8 shows you how you can use imaginary and actual address information to examine an EPF in memory.

EPF Mapping

The different parts of dynamic and registered EPFs are mapped to different locations in memory. PRIMOS maintains Descriptor Table Address Registers (or **DTARs**) for EPF memory addressing as follows:

- DTAR1 — shared memory segments
- DTAR2 — per-user memory segments (user assigned)
- DTAR3 — per-user memory segments (system assigned)

Refer to the *System Architecture Guide* for further details.

Both the pure and per-user parts of dynamic EPFs are mapped to unused segments in DTAR2 during program execution. The number of DTAR2 segments available for dynamic allocation can be set by the System Administrator. PRIMOS maps dynamic EPFs only to these dynamic segments, assuring that data in static segments is never corrupted by EPF memory usage.

Both the pure and per-user parts of registered EPFs are mapped to DTAR1 segments at registration time. At the same time, PRIMOS reserves virtual DTAR3 segment numbers to hold the per-user part of the EPF. PRIMOS maintains a list of DTAR3 segments reserved for each registered EPF. These DTAR3 segments are used when a user invokes the registered EPF. When a user invokes the EPF, PRIMOS copies the image of the per-user part in DTAR1 directly to these reserved DTAR3 segments. All references to addresses in the per-user part are adjusted to point to locations in the reserved DTAR3 segments. Therefore, unlike a dynamic EPF, the per-user part of a registered EPF is mapped to the same virtual location in every user's virtual address space.

EPF Sharing

PRIMOS shares registered EPFs automatically at registration time by placing shareable code and linkage in DTAR1 segments that are part of every user's address space. (The *–public* search rule must be set to support EPF sharing, as described in Chapter 6.)

PRIMOS shares dynamic EPFs using Virtual Memory File Access. Virtual Memory File Access (VMFA) provides two important benefits:

- Pure code can be paged directly from the file system copy.
- All users can share the same copy of the pure code.

For pure code, VMFA pages the physical memory copy of the EPF file image directly from the file system copy of the dynamic EPF. Because the EPF file image does not change, PRIMOS does not need to write it out to the paging disk. The physical memory copy is simply overwritten. PRIMOS can always page in a new copy directly from the file system.

PRIMOS uses the address translation mechanism to map the same physical memory copy of the pure code to each user's virtual address space. Once a dynamic EPF is mapped into one user's address space, the same physical memory copy is mapped to available segments in any subsequent user's virtual address space. Since the EPF file image contains the pure portion of the dynamic EPF, the same copy of the pure code is shared by all users who invoke this dynamic EPF.

Only the impure portion of a dynamic EPF is not shared in this way. After PRIMOS maps the EPF, it allocates space for per-user data and linkage from the dynamic segments in each user's address space. PRIMOS then initializes these areas by expanding the compressed description contained in the EPF file image. It changes imaginary addresses to actual addresses in the user's virtual address space. For example, IPs in the linkage/data segment that point to imaginary addresses are changed to point to the actual addresses in the user's virtual address space.

3 The Life of an EPF

All EPFs pass through a number of phases. Depending on the EPF type, these phases may occur at different times and in different order. Some phases only occur with certain EPF types. This chapter outlines the steps that EPFs go through as they are created and run, and tracks the life of some typical EPFs through all phases.

The Life of a Dynamic Program EPF

A dynamic program EPF typically goes through the following phases:

1. You write and compile or assemble the program.
2. You use BIND to build an EPF.
3. You invoke the EPF.
    - o PRIMOS maps the pure procedure portion of the EPF to memory.
    - o PRIMOS allocates the impure linkage/data portion of the EPF.
    - o PRIMOS initializes the impure linkage/data portion of the EPF.
4. PRIMOS calls the main entrypoint of the EPF, beginning execution.
5. PRIMOS resolves dynamic links encountered in the EPF during execution.
6. The EPF terminates, returning to its caller.
7. You or PRIMOS remove the EPF from memory.

The first time you invoke a dynamic program EPF during a terminal session, it must pass through all phases except the last one. On subsequent invocations, PRIMOS may be able to skip some or all of the mapping, allocation and initialization phases.

Compiling or Assembling

When you compile (or assemble in the case of PMA) your source code, you create an object code file, also referred to as a .BIN file. This .BIN file contains procedure text and linkage/data text. The procedure text consists of the program instructions and constant data. The linkage/data text consists of

- Static data that can be initialized when the program is invoked
- Entry Control Blocks (ECBs) for the main entrypoint and internal subroutines
- External linkage information such as IPs and dynts to external routines

The compiler can determine some of this linkage/data text. It establishes initial values for static data and most of the contents of the ECBs. However, the actual locations of the procedure and linkage frames remain unknown, and all external references remain unresolved.

The linkage part of the text includes information about unknowns that are resolved by linking the program with BIND. For example, compiling a call to an external routine, SUBR_A, generates linkage text that indicates the need for an indirect pointer (IP) to SUBR_A at a certain location in the linkage frame. When you link this program, BIND creates the IP.

Building With BIND

The BIND linker resolves the unknown addresses and external references in your object file. As you load object files, BIND builds up the procedure and linkage/data segments of your EPF. BIND places the procedure code in procedure segments and linkage and static data in the linkage/data segments. As each element is placed, BIND determines its imaginary address.

Once BIND has determined the imaginary address of an element, BIND is able to resolve references to the element. For example, once BIND has determined the imaginary addresses of a procedure's linkage and procedure frames, those addresses can be placed in the procedure's ECB.

External References: BIND maintains and updates a list of external references as you build your program. For example, if you load a module called MY_PROG that calls an external subroutine called SUBR_A, BIND adds SUBR_A to the list of unresolved references in MY_PROG's linkage.

BIND handles two types of external references:

- References to routines loaded during the BIND session
- References to routines in runtime libraries

If the referenced routine is in a module that you load as part of the BIND session, BIND handles the external reference by generating an imaginary address. That is, BIND creates an IP that points to the actual location of the subroutine's ECB in the linkage text. For example, if you load SUB_A.BIN (using the LOAD subcommand), BIND places the procedure and linkage of SUB_A in procedure and linkage segments. Included in SUB_A's linkage is SUB_A's ECB. BIND can now place the imaginary address of SUB_A's ECB in an indirect pointer (IP) in MY_PROG's linkage. This resolves MY_PROG's external reference to SUB_A.

If the referenced subroutine is in a runtime library, BIND handles the external reference by creating a dynt. During the BIND session, you can either explicitly load the appropriate binary library or you can issue the DYNT subcommand. The DYNT subcommand searches your binary libraries to resolve external references. For example, suppose MY_PROG calls an external subroutine, called SUB_B, that resides in a library EPF. You begin the BIND session by loading MY_PROG; BIND adds SUB_B to the list of unresolved references in MY_PROG's linkage. If you then give the subcommand DYNT SUB_B, BIND does two things:

- BIND places the character string "SUB_B" in MY_PROG's procedure text. (Since the string is a constant, it can go in a pure segment.)
- BIND creates a dynamic link in MY_PROG's linkage text. It sets the fault bit of the dynamic link to 1 (making it a faulted IP) and places the imaginary address of the character string "SUB_B" in this faulted IP.

This resolves the reference to SUB_B. BIND can accomplish the same result if you load the dynt with a binary library.

At the end of a successful BIND session, all previously unresolved references in your program have been resolved to either imaginary addresses or dynts.

Main Entrypoint: In addition to resolving references, BIND determines the entrypoints of your EPF. A program EPF has only one entrypoint. By default, BIND uses the first procedure loaded as the main entrypoint. You can specify a different entrypoint with the MAIN subcommand.

Program Invocation

You can invoke a program EPF in one of three ways:

- Directly from a command line: as a program (using the RESUME command), as a command, or as a function
- From a program that calls the EPF$RUN subroutine
- From a program that calls the CP$ subroutine

When you invoke a program EPF (using any of these methods), PRIMOS automatically performs several operations that prepare it for execution. These operations are normally invisible to the user. They include

- Pure procedure mapping
- Linkage/data allocation
- Linkage/data initialization

PRIMOS performs these operations by calling a series of subroutines. The specific subroutines responsible for each step are noted in the descriptions that follow. Normally you need not call these subroutines; PRIMOS automatically calls them you when you invoke a program EPF from the command line or with the EPF$RUN or CP$ subroutines. However, you can call these subroutines if you wish to perform any of the EPF mapping, initialization, and invocation steps individually. These EPF-related subroutines are documented in detail in *Subroutines Reference II: File System* and in *Advanced Programmer's Guide III: Command Environment.*

If the EPF has been invoked previously, some of these steps may already be complete. When you invoke an EPF, PRIMOS checks to see if a step may be omitted. In the case of a dynamic program EPF, the EPF may still be mapped as a result of a previous invocation. If this is the case, PRIMOS skips the mapping step.

Procedure Invocation: You should note the difference between program invocation as described above, and procedure invocation.

The basic functional unit of EPF organization is the **procedure**. A procedure includes executable code, linkage, and data. Every procedure has an entrypoint, defined in its linkage by an **Entry Control Block** (ECB), that allows the procedure to be called in an orderly way. A given EPF may consist of one or more procedures.

Note Procedures are functional units. The actual organization of an EPF in memory is by segments. Different parts of a procedure are placed in different segments, and given segment may contain code from many procedures. See Chapter 2 for a discussion of EPF organization.

Whether a procedure is contained in a program EPF or a library, PRIMOS invokes it in essentially the same way. Normally, procedures are invoked by executing a PCL instruction that addresses the first location in the procedure's ECB (or a faulted IP, which is resolved to point to the ECB). The PCL instruction allocates a stack frame for the called procedure, passes any arguments and causes execution to jump to the first executable instruction of the procedure. This sequence of events is called **procedure invocation**. (Procedures can also be called by various jump instructions without using an ECB or allocating a stack frame. These are called short calls and quick calls.)

Procedure invocation occurs in two circumstances:

- A running procedure calls a procedure that has been linked to it.
- A program EPF is invoked either from the command line or from another program.

When you invoke a program EPF, the EPF$ subroutines take care of mapping and initializing the EPF, process the command line, and finally invoke the EPF's starting procedure. In other words, the last step of program invocation is a procedure invocation.

The term **invocation** is also frequently used to mean the running of a procedure from the point when it is invoked to the point when it terminates and returns to the invoking procedure. In this sense, a given program invocation consists of one or more (usually many more) procedure invocations. A process typically consists of many program and procedure invocations.

Typically, a program EPF calls several subroutines, which in turn call other subroutines, and so on. All the subroutines called, both directly and indirectly, by a given program invocation are considered to be part of that invocation.

Pure Procedure Mapping

To map a dynamic EPF, PRIMOS allocates sufficient dynamic segments to hold all of the pure procedure segments. PRIMOS maps the imaginary segment number of each procedure segment (+0, +2, +4, and so on) to one of the allocated dynamic segments.

PRIMOS does not actually read the procedure text in from the file system at this point. Instead, PRIMOS pages the procedure text in from the file system during program execution. As the program executes, PRIMOS uses the virtual memory mechanism (VMFA) to page data from the file containing the EPF directly into the allocated procedure segments. PRIMOS automatically pages in the procedure text during execution on an as-needed basis. Because the virtual memory mechanism does not allow segments mapped in this way to be modified, PRIMOS sets access to these segments so that they cannot be written by the user or the program. Virtual memory mapping is described in detail in Chapter 8.

The subroutine EPF$MAP carries out EPF mapping.

Linkage Allocation

PRIMOS also allocates sufficient dynamic space to hold all of the linkage/data and any impure procedure segments required by the EPF. PRIMOS sets the access to these segments so that they can be written to by the user or by the program. The subroutine EPF$ALLC carries out linkage allocation.

Linkage Initialization

Once space for the linkage segments has been allocated, PRIMOS reads in the linkage description from the EPF file and expands it to create the contents of the linkage segments. PRIMOS converts imaginary addresses into actual addresses in virtual memory. PRIMOS sets the IPs for each internal subroutine to contain the virtual address of that subroutine. PRIMOS sets the faulted IPs for each external routine to contain the virtual address of the character string that hold the dynt name.

PRIMOS also copies any impure procedure code into impure segments during this phase.

The subroutine EPF$INIT carries out linkage/data initialization.

Entrypoint Invocation

At this point, the EPF is ready to execute. PRIMOS executes a PCL instruction to the ECB of the main entrypoint, and the EPF code begins to execute.

The subroutine EPF$INVK invokes the main entrypoint of a program EPF.

Resolving Dynamic Links

PRIMOS resolves dynamic links (snaps dynts) as it encounters calls to external routines during program code execution. The first time PRIMOS executes a call to a particular external routine, it resolves the dynamic link. Subsequent calls in that program to the same external routine often use the same resolved dynamic link.

PRIMOS recognizes an unresolved dynamic link when it encounters a faulted IP. The faulted IP generates a fault condition. This brings the dynt snapping mechanism into play, as described in Chapter 2.

When the dynt is snapped, PRIMOS replaces the faulted IP with an IP containing the actual address of the routine in virtual memory. PRIMOS then reissues the call to the external routine.

Subsequent calls to the routine through the same IP are executed without producing a fault condition. Calls to the same routine from other parts of the program may be made through different faulted IPs. In this case, these dynts still need to be snapped.

In practice, few programs ever resolve all of their dynamic links. PRIMOS only resolves the dynamic links of those external routines that are actually called. Your program may contain references to external routines that are not called during every execution of the program. Dynamic links to those routines remain unsnapped during invocations when the routines are not referenced.

EPF Termination and Reinvocation

Normally, a program EPF terminates by returning to the routine that invoked it (the EPF$INVK subroutine).

Note An EPF may also terminate and return to command level by calling the EXIT routine. This is not usually recommended since this defeats some features of the flexible EPF command environment.

When an EPF terminates, PRIMOS marks the linkage area used by that invocation for reinitialization. PRIMOS does not deallocate the linkage area. If the EPF is subsequently reinvoked from the same command level, the linkage area does not need to be reallocated, just reinitialized. When the linkage area is reinitialized, only program data and IPs are actually reset, saving startup time.

PRIMOS resets all faulted IPs in order to be sure that any program-class libraries called are properly reinitialized when the program is run again. See the discussion of library initialization in Chapter 5.

PRIMOS deallocates an EPF linkage area only when the command level that invoked it is released or the EPF is removed from memory. When the same EPF is invoked at more than one command level, PRIMOS allocates a separate linkage area for each command level. Only the linkage area for the current command level is deallocated unless the other command levels are released. By keeping linkage areas independent, PRIMOS assures that suspended program invocations are not affected by the current invocation.

When a program EPF terminates, PRIMOS also deallocates dynamically allocated memory acquired during execution.

Note If your program EPF is called as a command function, it normally allocates space for the returned string using the ALS$RA or ALC$RA subroutines. It is the responsibility of the calling program to deallocate this space using the FRE$RA subroutine. If you call an EPF as a command function from another program, you should have the calling program deallocate the returned string space. Refer to the *Advanced Programmer's Guide III: Command Environment* for further details.

Removing an EPF From Memory

PRIMOS tries to keep dynamic program EPFs mapped to memory after they terminate. PRIMOS places information about a terminated EPF in a data structure called the **EPF cache**. While the EPF is in the cache, it remains mapped to memory. If the EPF is reinvoked while in the cache, it need not be remapped. This helps reduce startup time.

EPFs that have terminated but remain mapped are listed as `(not active)` by the LIST_EPF command.

When the number of EPFs mapped to memory becomes too large, PRIMOS removes the oldest inactive EPF from the cache and unmaps it. If the EPF's linkage/data segments are still allocated, PRIMOS deallocates them at the same time.

You can explicitly remove inactive EPFs with the REMOVE_EPF command. The subroutine EPF$DEL also handles EPF removal.

The Life of a Dynamic Library EPF

A dynamic library EPF must pass through the same stages as a program EPF, although the order and details differ in some cases. This section explains the important differences.

- A library EPF is invoked only when PRIMOS snaps a dynt to it.
- PRIMOS maps the pure portion of a dynamic library EPF when it encounters the EPF name in an ENTRY$ search list. Once a library has been mapped by a user process, it remains mapped unless the user's command environment is reinitialized or the library is explicitly removed. During subsequent dynt snapping, PRIMOS can skip the mapping phase for EPFs already mapped.
- PRIMOS allocates and initializes the impure portion of a library EPF when it snaps a dynt to it. PRIMOS decides whether it needs to carry out these phases depending on the class of the library and whether the calling program or process has previously snapped another dynt to the same library.

Invoking and Mapping

A dynamic library EPF is invoked when a running program calls one of its entrypoints.

PRIMOS must map a dynamic library EPF to search its entrypoints. The first time a program EPF calls a routine in a library EPF, PRIMOS maps the library EPF. PRIMOS then searches the entrypoints of that library EPF, attempting to locate the called routine. If PRIMOS finds the called routine, it snaps the dynamic link, then reissues the call to that routine. Therefore, unlike a program EPF, a dynamic library EPF is always mapped by the time it is actually invoked by a call from a running program.

PRIMOS finds the correct library EPF by examining the libraries listed in the user's ENTRY$ search list. PRIMOS searches the listed libraries until it finds one that contains the correct entrypoint. In order to search each library's entrypoints, PRIMOS needs to map each library (unless the library is already mapped). Typically, dynamic linking results in the mapping of many library EPFs as PRIMOS searches for routines. Dynamic links may not have been snapped to most of these libraries, but as long as they remain mapped, they need not be remapped during subsequent dynamic linking.

Linkage/Data Allocation and Initialization

Each time the dynamic linking mechanism snaps a dynt to a library EPF, PRIMOS determines whether it needs to allocate and initialize the library's linkage/data segments. This decision depends on whether the EPF is a program-class or process-class library and whether it is already in use by the same program or process. (Whether a library EPF is a program-class or process-class library is established by the LIBMODE subcommand of BIND. Refer to the *Programmer's Guide to BIND and EPFs* for details.)

Program-class Libraries: PRIMOS maintains a different copy of the linkage/data segments of a **program-class library** for each active program invocation that calls that library. When a program that calls the library terminates, the library linkage/data segments used by that invocation are marked for reinitialization. These segments can be reinitialized and used by a subsequent program invocation that calls the library.

PRIMOS initializes the linkage/data portion of a program-class library the first time the library is called by a given program invocation. When snapping a link to a program-class library, PRIMOS checks to see whether the current program invocation has already snapped another link to the same library. If it has, then the linkage/data segments have already been allocated and initialized, and PRIMOS need not repeat these steps.

Process-class Libraries: PRIMOS maintains only one copy of the linkage/data area of a **process-class library** for each process that calls the library. PRIMOS allocates and initializes this linkage/data area the first time a routine in the EPF library is called by a given user process. When snapping a link to a routine in a process-class library, PRIMOS checks to see whether the library has already had its linkage/data allocated and initialized by the same process. If it has, PRIMOS can skip these steps.

The linkage/data portion of a process-class library is deallocated only when one of the following happens:

- The user logs out.
- The user explicitly removes the library with the REMOVE_EPF command.
- The user's command environment is reinitialized, either by the INITIALIZE_COMMAND_ENVIRONMENT (ICE) command or by an error condition.

Calling Routines From a Library EPF

A running program calls a routine in a library EPF. As that library EPF routine executes, it may itself call routines in other library EPFs. The library EPF calls the routine, encounters a faulted IP, and resolves the dynamic link. The mechanism is the same as the one used by program EPFs to resolve dynamic links.

As these dynts are snapped, PRIMOS needs to decide whether to initialize the called libraries. When making this decision, PRIMOS considers the calling library to be part of the program invocation that called it, either directly or indirectly. If a called program-class library has already been referenced by the same program invocation, PRIMOS does not reinitialize its linkage/data.

Termination and Removal

When the program invocation that calls a library terminates, PRIMOS decides how to dispose of the library. If the library is not linked to another active program, the library is considered inactive. PRIMOS disposes of the library's linkage/data area according to the library type:

- When a program-class library terminates, the linkage/data area for the terminated program invocation is marked for reinitialization. The area remains allocated and available for use by another program invocation.
- When a process-class library terminates, PRIMOS leaves the linkage/data area allocated for the process untouched. Other program invocations that call the library from the same process use the same linkage/data area without reinitializing it.

Both program-class and process-class libraries remain mapped after the program invocations that called them have terminated. You can remove inactive libraries of both types from your address space with the REMOVE_EPF command.

The Life of a Registered EPF

A registered program EPF goes through a somewhat different set of phases. You write, compile (or assemble), and use BIND to build the EPF, much as you would with a dynamic EPF. Then

1. The System Administrator registers the EPF. At this point PRIMOS carries out several steps:

   A. It maps the shared portion to shared memory.

   B. It creates an initialized copy of the per-user portion in shared memory.

   C. It snaps dynts in the shared portion.
   D. It registers the EPF in a table of registered EPFs.

2. You invoke the EPF from the command line or it is invoked from a running program. PRIMOS then performs the following steps:

   A. It maps the initialized copy of the per-user portion to PRIMOS per-user segments.
   B. It invokes the main entrypoint, beginning execution.
   C. It snaps per-user dynts as they are encountered during execution.

3. The EPF terminates, returning to its caller.
4. The shared portions of a registered EPF remain mapped to memory until the EPF is unregistered by the System Administrator.

For more information on registered EPFs, see Chapter 6.

4

Program EPFs

A program EPF is an executable object designed to be called from the command line or explicitly from another program. It contains a single main entrypoint that is invoked when the program is called.

A program EPF can be a dynamic EPF or a registered EPF. Program EPFs are distinguished from library EPFs, as shown in Figure 4–1. A program EPF has a single entrypoint; a library EPF may have multiple entrypoints. You can invoke a program EPF directly from the command line or indirectly via a call to a Prime-supplied subroutine. You never directly invoke a library EPF; you execute routines within a library EPF by calling them from a program EPF. Library EPFs are further described in Chapter 5; special considerations for registered EPFs are described in Chapter 6.

Dynamic Program EPF • Single entrypoint • Direct invocation	Dynamic Library EPF • Multiple entrypoints • Indirect invocation
Registered Program EPF • Single entrypoint • Direct invocation	Registered Library EPF • Multiple entrypoints • Indirect invocation

Figure 4–1. Comparison of Entryname and Invocation Properties of Program EPFs and Library EPFs

A program EPF can be invoked in three ways:

- As a program
- As a command
- As a command function

You invoke a program EPF as a program by typing the command line

```
RESUME program_name
```

You invoke a program EPF as a command by typing the program name. You invoke a program EPF as a command function by typing the program name in square brackets as

```
[program_name]
```

The function returns a string value.

You may also call a program EPF in any of these forms from a running program. A running program calls a program EPF by using either the CP$ subroutine or the EPF$ subroutines. The CP$ subroutine calls a program EPF by passing a command line to the command processor. You can call both dynamic program EPFs and registered program EPFs using the CP$ subroutine. The EPF$ subroutines directly invoke a program EPF, giving you greater control over EPF mapping and execution. These subroutines are documented in the *Subroutines Reference II: File System* and their use is further detailed in the *Advanced Programmer's Guide III: Command Environment.* You can call dynamic program EPFs (and registrable program EPFs) using the EPF$ subroutines; you cannot call a registered program EPF using these subroutines.

Whether a dynamic EPF can be invoked as a program, command, or command function depends on how the calling sequence is coded and how the EPF is installed.

Coding and Compiling

When you code a program EPF, you specify the calling sequence of its main entrypoint. Prime recognizes five standard calling sequences. Which calling sequence you select determines how the program EPF can be invoked. If you write a program EPF that will be invoked as a program, you normally need not concern yourself with the calling sequence. If you write a program EPF that will be invoked as a command, you should code for command line argument passing and a severity code return value. If you write a program EPF that will be invoked as a command function, you should code for the string value return mechanism. These calling sequences are described in detail in the *Advanced Programmer's Guide III: Command Environment.*

Aside from calling sequence specification, there are no important restrictions on program EPF coding. In order to take full advantage of the EPF mechanism, program EPFs should be pure code. See Appendix A for information on how to code program EPFs in PMA.

Compiler Options

EPFs must be compiled in V-mode or I-mode. Do not use R-mode. You can use IX-mode if your compiler supports this extension to I-mode. IX-mode considerations are further described in Chapter 5.

It is recommended that you do not use the –PBECB compiler option. This option provides greater locality of reference by placing ECBs in the procedure text. Since PRIMOS must modify ECBs during program execution, PRIMOS must store these ECBs and their associated procedure text in impure code segments that cannot be shared. This typically reduces the performance of an EPF. However, in some very large programs, the benefits of improved locality of reference may exceed the performance costs of these impure code segments.

Note PRIMOS automatically places some ECBs in shared procedure segments when you use BIND to build a registered EPF. The PBECB compiler option cannot be used to enhance this aspect of registered EPFs.

Linking With BIND

When you link a program using BIND, it creates a dynamic program EPF as the default type. BIND creates a dynamic program EPF unless you specifically request another type. Linking dynamic program EPFs is usually very straightforward. For a complete guide, see the *Programmer's Guide to BIND and EPFs.*

Note Additional information you need if you want to create registered EPFs is given in Chapter 6. Even if you intend to create a registered EPF, you should still understand the material in this chapter. Most of the material in this chapter is applicable to both kinds of program EPFs. Furthermore, the recommended procedure for building registered EPFs is to build and test them first as dynamic EPFs.

Defining the Main Entrypoint

The main entrypoint of an EPF is determined by BIND. The BIND linker has a MAIN subcommand for this purpose. When you use BIND to build the program, the main entrypoint is one of the following:

- The procedure specified by the MAIN subcommand
- If you do not issue a MAIN subcommand, the first procedure linked during the BIND session

Generally, the first procedure in the source code becomes the first procedure in the object file and, therefore, the first procedure linked by BIND. If you do not specify the MAIN subcommand, this first procedure becomes the main entrypoint by default. Usually, when you build a program you allow BIND to choose the first procedure as the main entrypoint. However, you can always override this selection by issuing the MAIN subcommand.

Note that the main entrypoint is not necessarily determined by the syntax of the main procedure declaration in the language you use. Each language has its own conventions for defining a main procedure:

- In F77 the main procedure customarily begins with the PROGRAM *prog_name* statement.
- In PL/I the main procedure begins with *prog_name* PROC OPTIONS(MAIN).
- In C the main procedure is called main( ).

Note Appendix A shows you how to write the main entrypoint of a PMA program that is to be executed as an EPF.

However, none of these statements guarantees that the declared procedure is considered the main procedure by BIND. This depends on the order in which procedures are actually linked or on the MAIN subcommand of BIND. For example, you can write a C program that declares two procedures:

```
main()
{
      .
      .
      .
}

subroutine_a()
{
      .
      .
      .
}
```

Normally, `main` is the main entrypoint. However, if you give the command

```
MAIN subroutine_a
```

during the BIND session, BIND marks *subroutine_a* as the main entrypoint. This is not recommended. In fact, if you fail to use the correct syntax for the main entrypoint, your program may not execute properly. But the possibility of building a program this way does emphasize the fact that BIND rather than the compiler ultimately determines the main entrypoint.

The MAIN subcommand is useful if you build your programs from many separate modules. You can link modules in any order and then explicitly declare the main entrypoint with the MAIN subcommand.

Installation

To install a dynamic program EPF, you place the .RUN file in an appropriate directory, set the access rights to the EPF, and (if required) set the system or individual user's COMMAND$ search rules.

To invoke a program EPF as a program, you must place the EPF in a directory to which all of the program's potential users have the necessary access rights. To execute an EPF on a local disk, a user must have Read or Execute rights to the EPF and Use rights to its directory. To execute an EPF on a remote disk, a user must have Read rights to the EPF and Use rights to its directory. Access rights are documented in the *PRIMOS Users Guide.*

If the EPF is to be called as a command or command function, you must also include the name of the EPF's directory in each potential user's COMMAND$ search rules. There are two ways to do this:

- Place the EPF in a directory listed in the system's COMMAND$ search list.
- Place the name of the EPF's directory in the COMMAND$ search list of every user who will invoke the program EPF as a command or command function.

Note that a user's search lists are automatically reset to the system's search lists every time the process is reinitialized (for example, by logging in or issuing the ICE command). The SET_SEARCH_RULES and LIST_SEARCH_RULES commands are described in the *PRIMOS Commands Reference Guide*; the search rules facility is described in the *Advanced Programmer's Guide II: File System.*

5 Library EPFs

A library is a collection of one or more (usually many) executable routines. When a program EPF calls an external routine, PRIMOS locates the executable code for that routine in a library. A library can, in turn, call routines in other libraries. This chapter describes how to create and maintain library EPFs. It also describes libraries that are not EPFs.

A library EPF can be a dynamic EPF or a registered EPF. Library EPFs are distinguished from program EPFs as shown in Figure 5–1. A program EPF has a single entrypoint; a library EPF may have multiple entrypoints. You invoke a program EPF directly; you invoke a library EPF indirectly by issuing calls to its routines. You can call routines in a library EPF from a program EPF or from another library EPF. Program EPFs are further described in Chapter 4; special considerations for registered EPFs are described in Chapter 6. Library EPFs are divided into two classes: program-class and process-class; library classes are described in this chapter.

Dynamic Program EPF • Single entrypoint • Direct invocation	Dynamic Library EPF • Multiple entrypoints • Indirect invocation • Program-class or process-class
Registered Program EPF • Single entrypoint • Direct invocation	Registered Library EPF • Multiple entrypoints • Indirect invocation • Program-class or process-class

Figure 5–1. Comparison of Properties between Program EPFs and Library EPFs

The Library Mechanism

A library EPF is a collection of routines, each routine having its own entrypoint. In general, libraries contain many routines related to a given function, compiler, application, or product. This means that even though a program may call many routines, BIND can usually resolve these calls by linking to a small number of libraries. Therefore, storing similar routines in a library EPF saves a great deal of linking effort, when compared with linking to individually-stored routines.

The library mechanism functions in two stages:

- At link time
- At run time

At link time, you create the dynts that the dynamic linking mechanism uses at run time. In most cases, you do this with BIND by linking to **binary libraries** that contain code to create the dynts. You can also create dynts using the DYNT subcommand of BIND.

At run time, the dynamic linking mechanism resolves these dynts to link your program EPF to routines contained in **runtime libraries** of various types. One type of runtime library is library EPFs. (Registered EPFs resolve some dynts at registration time, before the program is actually run.)

The library mechanism thus uses two types of libraries:

- Runtime libraries
- Binary libraries

Runtime Libraries

A **runtime library** contains executable versions of routines. A runtime library can be a library EPF, a shared static-mode library, or a PRIMOS direct entry. A program EPF can call routines in a runtime library; a routine in a library EPF can call other routines in other runtime libraries.

A call to a routine in a runtime library is performed by establishing (at BIND time) and resolved (at runtime) dynamic links in the calling program. The code in runtime libraries never becomes a part of the calling program. Instead, BIND incorporates a dynamic link to each called routine in the calling program. The next section, Binary Libraries, shows how BIND does this.

EPFs can link dynamically to three types of runtime libraries:

- Shared static-mode libraries
- PRIMOS direct entries
- Library EPFs

Each type of library consists of a collection of routines and a list of entrypoints. Each type is organized and stored differently, but PRIMOS links to them dynamically in essentially the same manner. It searches the entrypoint list for the name of the called routine. If it finds the name, it uses a pointer supplied in the entrypoint list to resolve the dynamic link to the actual runtime code of the routine. Chapter 2 explains the dynamic linking process in detail.

Shared Static-mode Libraries: Shared static-mode libraries contain static-mode routines that are loaded into shared segments. All EPFs that call routines in shared static-mode libraries execute the same shared copy. As of Rev. 23.0, registered library EPFs have replaced most shared static-mode libraries supplied by Prime (refer to Appendix D for details). The remaining shared static-mode libraries are mainly used by shared static-mode products.

PRIMOS Direct Entries: A PRIMOS direct entry is a routine that is actually a part of the PRIMOS operating system. Because these routines are part of the single shared copy of PRIMOS, each EPF that calls them uses the same shared copy.

You can determine if a called routine is a PRIMOS direct entry when you use BIND to build the calling program. Use the MAP subcommand of BIND to observe when references are resolved. PRIMOS direct entries are those references that BIND resolves when you issue the LIBRARY (LI) subcommand with no specified library name.

Library EPFs: Library EPFs are EPF runfiles that contain a collection of routines. You can create your own library EPFs using the BIND linker. Prime also supplies many library EPFs that contain routines called by various Prime products.

Binary Libraries

A **binary library** is a collection of object code modules kept together in one .BIN (binary) file. You link these modules into your runfile using BIND and they become part of the runfile. Note the distinction between runtime libraries and binary libraries: the code in a runtime library is accessed by dynamic links from your program; the code in a binary library is actually copied into your program.

Binary libraries may contain the compiled code of subroutines. In this case, BIND copies the binary code of the called subroutines into your program's runfile. The called subroutines become part of your program.

However, most binary libraries do not contain the actual code of subroutines. Instead, many binary libraries contain the object code for dynamic links to routines in runtime libraries. BIND copies the dynamic links for called routines into your program's runfile. These dynamic links become part of your program.

Most Prime-supplied binary libraries contain code for creating dynts to routines in runtime libraries. When you link to these binary libraries, PRIMOS does not put any actual routine code in your runfile. Instead, it puts in the dynts that it needs to link to these routines at runtime.

You can use the LIBRARY subcommand of BIND to link such binary libraries. This is one way to establish dynamic links in your runfile. You can also use the DYNT subcommand of BIND to create individual dynamic links. Binary libraries simplify program linking by making it unnecessary to create each dynt individually. The *Programmer's Guide to BIND and EPFs* describes these BIND subcommands in greater detail.

The dynts in Prime-supplied binary libraries do not necessarily point to routines contained in any one runtime library. Instead, they point to routines in a variety of runtime libraries, including both PRIMOS entrypoint libraries (direct entries) and library EPFs. A specific runtime routine may be referred to by dynts in several binary libraries. These binary libraries provide a flexible interface at link time between your program and the runtime libraries. Because a single binary library can contain dynts to routines in several runtime libraries, you can usually resolve all external references in your program by linking to one or two binary libraries.

Creating Binary Libraries of Dynts: After you create a library EPF with BIND, you can use EDIT_BINARY to create a binary library that contains dynts to the routines in that library EPF. You can then use BIND to copy this binary library into programs, creating dynamic links to your library EPFs, just as you would with Prime-supplied libraries.

You link to Prime-supplied binary libraries by using the LIBRARY subcommand of BIND. Prime-supplied binary libraries reside in the top-level directory LIB. The command LIBRARY *library_name* is functionally equivalent to the command LOAD LIB>*library_name.*

You link to your own binary libraries by using the LOAD subcommand of BIND, or by using the LIBRARY subcommand and supplying a complete pathname. You can request that your System Administrator place your own binary libraries in the LIB directory.

Normally, binary libraries contain code that instructs BIND to link in only the modules that it needs to satisfy unresolved references. When you link to a binary library, BIND searches the library for dynts or routines that satisfy unresolved references and copies them into the runfile.

ENTRY$ Search List

When the dynamic linking mechanism attempts to resolve a dynamic link, it uses your ENTRY$ search list to look for the required routine. An ENTRY$ search list contains a set of search rules that give the order in which libraries should be searched to find a routine. Each rule is an instruction to search one library or type of library.

The ENTRY$ search rules may include the following:

–PRIMOS_DIRECT_ENTRIES	Tells PRIMOS to search for the routines among PRIMOS direct entrypoints. This is always the first rule.
–STATIC_SHARED_LIBRARIES	Tells PRIMOS to search shared static-mode libraries.
EPF_pathname	Tells PRIMOS to search for entrypoints in the named library EPF.
–PUBLIC	Tells PRIMOS to search registered library EPFs.

PRIMOS searches the libraries in your ENTRY$ search list in the order in which they are listed until it finds a library that contains the requested entrypoint.

PRIMOS provides a set of system default ENTRY$ search rules. These system default search rules normally contain entries for all Prime-supplied runtime libraries and any other libraries that the System Administrator has chosen to install on the system.

An individual user may modify or replace the default search rules by creating a private ENTRY$ search list. If the applications you run make especially heavy use of certain runtime libraries, you may want to edit your search rules to put the names of those libraries close to the top. This can make dynamic linking more efficient. You may also wish to add your own user-written library EPFs to your ENTRY$ search list.

However, if you link a program with routines in a user-written library EPF, you must be sure that every user that runs that program is using an ENTRY$ search list that contains a search rule for that library. When a running program attempts to call a routine in a runtime library not included in your ENTRY$ search list, the program fails, generating a LINKAGE_FAULT$ error.

Note that a user's search lists are automatically reset to the system's search lists every time the process is reinitialized (for example, by logging in or issuing the ICE command). The SET_SEARCH_RULES and LIST_SEARCH_RULES commands are described in the *PRIMOS Commands Reference Guide*; the search rules facility is described in the *Advanced Programmer's Guide II: File System.*

Using the Library Mechanism

EPFs give you complete access to the library mechanism. You can easily create your own library EPFs with BIND. You can also use EDIT_BINARY to create your own binary libraries containing dynts to the routines in your library EPFs. The remaining sections of this chapter show you all the steps you need to follow to create a library EPF and make it available for use.

1. Coding and compiling
2. Determining library class
3. Determining library entrypoints
4. Building a library with BIND
5. Creating a binary library
6. Installing the libraries

Chapter 6 shows you how to create a registered library EPF.

Coding and Compiling

You can write a library in any high-level language or PMA.

Coding Guidelines

The library source code is a collection of subroutines containing no main program. In PL/I you write a series of procedures with no program procedure. In FORTRAN 77 you create a set of subroutines or functions. In C you write a set of functions with no main function.

Routine Names: Usually you use the name you give a routine in the source code as the library entryname by which the routine is called from other programs. For example, suppose you define a FORTRAN 77 routine called SUMSQ using the statement

```
SUBROUTINE(S,N) SUMSQ
```

The process of building a library that contains SUMSQ is simpler if you use the same name in the library's entrypoint list. Some routine names are reserved for use by PRIMOS. The section, Determining Library Entrypoints, discusses library entryname conventions. You can simplify the process of building your library with BIND if you choose routine names according to these conventions.

Other Coding Guidelines: Routines can use any calling sequence. Declare parameters and return values according to the conventions of the language you are using. Library EPF routines differ from the main entrypoint of a program EPF in this respect. Main entrypoint calling sequences must follow specified formats, described in the *Advanced Programmer's Guide III: Command Environment*. Library EPF routine calling sequences impose no such requirements.

The nature of your routine code may affect the class requirements of your library. The next section explains library classes and gives coding guidelines.

Language Considerations: When choosing a programming language and designing calling sequences for routines in a library EPF, consider the requirements of the programs that will call those routines. It is generally easier to call a routine written in the same language as the calling program, since calling conventions and data types are the same. If a routine is to be called from several languages, try to choose compatible data types. For example, a PL/I routine that requires a CHAR (*n*) VARYING parameter is more difficult to call from C than a routine that requires a CHAR (*n*) parameter, because C contains no equivalent to the CHAR (*n*) VARYING data type.

Note If you plan to register your library EPF, make sure that the language and compiler version you are using support registration. See Chapter 6.

You should also remember that C language subroutines handle parameters by value, while FORTRAN 77 and PL/I handle them by reference. Subroutines written in FORTRAN 77 and PL/I can treat parameters as both input and output values in the same way that Prime-supplied routines do. C language subroutines must receive pointers to output parameters. If you write library subroutines in C, you should make programmers who use them aware of this difference.

If you are coding in PMA, see Appendix A, which gives general guidelines for writing EPF code in PMA.

Compiling

If your subroutines are written in a high-level language, compile your source using one of the Prime compilers. The compiler must generate V-mode or I-mode object code. If your subroutines are written in PMA, assemble the source with the PMA assembler. PMA code must contain the SEG or SEGR pseudo-operations, as described in Appendix A.

As with program EPFs, do not use the –PBECB compiler option. For more information, refer to Chapter 4.

Determining Library Class

When you build your library EPF using BIND, use the LIBMODE subcommand to specify whether it is to be a program-class library or a process-class library. You must specify a library class for all library EPFs, both dynamic library EPFs and registered library EPFs. This section shows you how to determine which class to use when building a library EPF.

Library Initialization

A library's class determines when and how often the library is initialized. Initialization of an EPF sets the initial values for various elements in the linkage/data segments. When PRIMOS initializes a linkage/data segment, it resets dynts to their unsnapped state and writes initial values to locations that contain static data. If an EPF contains impure procedure code, PRIMOS also copies the initial code values to the segments allocated for impure code.

Initialization may occur several times during the life of an EPF, depending on the type of the EPF. For dynamic EPFs, PRIMOS initializes the following:

- A **program EPF** each time it is invoked
- A **program-class library EPF** the first time it is called by a given program invocation
- A **process-class library EPF** the first time it is called by a given process

For registered EPFs, PRIMOS initializes the following:

- The **shared linkage/data** of the EPF at registration time.
- The **per-user linkage/data** of a registered EPF in the same way it initializes a dynamic EPF. The per-user linkage/data of a registered program EPF or a program-class registered library EPF is initialized once per program invocation. The per-user linkage/data of a process-class registered library EPF is initialized once per process.

The discussion that follows applies both to dynamic EPFs and to the per-user parts of registered EPFs.

When you create a library EPF you can choose to make it either a program-class or process-class library. In general, process-class libraries are preferable because they minimize initialization overhead. However, in certain circumstances you must create a program-class library. This depends on two issues:

- Whether the routines in the library EPF call routines in program-class libraries
- How the routines in the library EPF initialize and handle static data

Library-class Mixing

The following rules reflect the way PRIMOS initializes called libraries during program execution.

- A program-class library routine can call routines in another program-class library.

- A process-class library routine can call routines in another process-class library.
- A program-class library routine can call routines in a process-class library.
- A process-class library routine *cannot* call routines in a program-class library.

PRIMOS decides whether to initialize a called library when it snaps a link to the library. If the library is a program-class library, PRIMOS initializes it if it has not been previously called during the current program invocation. If the library is a process-class library, PRIMOS initializes it if it has not been previously called by the same process.

Because PRIMOS only makes this decision when it snaps a dynt, it cannot initialize a called library more often than it snaps dynts to the called library. Since dynts in the calling EPF remain snapped until the calling EPF is reinitialized, PRIMOS cannot be sure of initializing a called library any more often than it initializes the calling EPF. Therefore, process-class libraries, which are initialized once per process, should not call program-class libraries, which must be initialized once per program invocation. If PRIMOS detects a violation of this rule during dynamic linking, it generates a LINKAGE_FAULT$ error.

Therefore, any library routine that calls routines in program-class libraries must itself reside in a program-class library.

Language-directed I/O

Prime-supplied languages use program-class libraries to handle language I/O. Therefore, routines that use language-directed I/O must be in program-class libraries. Some language-directed I/O statements are shown in Table 5–1.

Table 5–1. Language-directed I/O Statements

Language	*Statements*
C	printf, getc, fopen
FORTRAN 77	READ, WRITE, OPEN, CLOSE
PL/I	put, get, open, close

Static Data Usage

A routine that use static data may also have to be placed in a program-class library. This depends on how the static data is used and how you intend the routine to function. This section gives some guidelines that you can use to determine whether static data usage requires that the routine be placed in a program-class library.

In general, compilers create non-static variables by default. Non-static variables are allocated and initialized at runtime in such a way that their lifespan is strictly limited to specific portions of the invocation, such as single subroutine invocation.

Static variables are allocated and initialized only as often as PRIMOS initializes the EPF. Therefore, the lifespan of static data in an EPF library depends on the library class. Static data values in a program-class library are maintained for a single program invocation. Static data values in a process-class library survive as long as the invoking process.

Static storage places data in fixed locations in the linkage/data area of an EPF. The precise scope and lifespan of static variables varies from compiler to compiler, but you typically use static storage to maintain the values of variables across calls to a procedure. Static data include

- Variables declared as static or static external in PL/I and C
- Variables declared with a SAVE, DATA, or COMMON statement in FORTRAN 77. (If the program is compiled with the -SAVE option, all variables are static.)
- PMA variables created using the COMM and EXT pseudo-operations or the LINK pseudo-operation followed by instructions such as DATA, OCT, DEC, BSS, BSZ, ECB, IP, and the like.

It is generally considered good programming practice to limit the use of static data in order to keep the behavior of your procedures consistent from invocation to invocation. If variable values are not maintained across calls, then you can be sure that values left over from one invocation cannot affect a procedure's behavior on subsequent invocations. This makes your procedures more robust and portable and allows you to program in a strictly modular way.

Note Even when you want to maintain variable values across calls, you can often avoid using static storage by passing the values back to the calling program and having the calling program maintain them.

Process-class Libraries: In order to minimize initialization overhead, process-class libraries are generally preferable. However, when using process-class libraries, you must be sure that static data values generated during one program invocation do not adversely affect the library's behavior during subsequent program invocations.

You can use process-class libraries explicitly to maintain static data values from program invocation to program invocation. Chapter 7 shows you how to make use of this feature to share data among programs within a single user process. Remember, however, that such use of static data may make your subroutines less easily portable among applications. You may be able to use other mechanisms, such as global variables, to maintain data values from program invocation to program invocation without sacrificing portability.

Program-class Libraries: Program-class libraries generate more initialization overhead, but insure that subroutine behavior does not vary from one program invocation to the next. They are useful when you choose to maintain static data values across calls during a single program invocation, but do not want these values to affect subsequent program invocations.

The following PL/I example illustrates the importance of correctly choosing the library class when library subroutines use static data. The sample routine uses static storage to maintain data across calls. It keeps a running average of a stream of numbers:

```
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 */
```

If this routine is placed in a process-class library, it can only keep the running average of a single stream of numbers for each process. For example, suppose that the program STATS calls this subroutine to calculate the average of a stream of 15 numbers, resulting in a value of 27. If you subsequently reinvoke STATS during the same terminal session, count is still equal to 15 and total equal to 27. Any new number stream is treated as a continuation of the previous one. The only way to avoid this behavior is to reinitialize your command environment between invocations of STATS.

If the same subroutine is placed in a program-class library, then subsequent invocations of STATS cause the values of count and total to be reinitialized. Each invocation of STATS can then use the AVERAGE subroutine to calculate the average of a different stream of numbers.

Note, however, that the utility of the routine is still quite limited by its use of static data. It can still only calculate the average of a single stream of numbers within a given program invocation since total and count maintain their values across calls throughout the program invocation.

You can make the routine much more flexible by rewriting it to eliminate the static data and pass the running values back to the calling program:

```
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 */
```

In this case, the calling program is responsible for maintaining the running value and the number count. There are other ways to handle this problem without eliminating static storage and requiring the calling program to maintain variable values across calls, but they can require careful management of the static data area.

Storage Allocation Issues

Program-class and process-class library EPFs differ in how they allocate and deallocate stack space.

50 Series architecture allows the dynamic allocation of stack space during a 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 PL/I

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 is not 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. Therefore, PRIMOS must distinguish between a program-class library EPF and a process-class library EPF when allocating memory.

A program-class library EPF acquires dynamically allocated memory from the **program-class storage** pool used by program EPFs. PRIMOS automatically deallocates this memory when a program terminates.

A process-class library EPF acquires dynamically allocated memory from a special memory pool, called **process-class storage**. To allocate memory from this pool, you must issue the LIBRARY PROCESS_CLASS subcommand when you use BIND to build the library. No memory from this pool is ever explicitly deallocated by PRIMOS except during logout and command environment initialization.

If you build a process-class library EPF without the LIBRARY PROCESS_CLASS subcommand, then any language-driven allocation, either explicitly via statements such as ALLOCATE in PL/I, or implicitly via compiler-generated allocation for temporary storage, fails when the library EPF executes. The failure is 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 routine if that routine 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 routine unless the routine stops using the address when it returns to its caller.

In general, a pointer to object A should never be passed to routine B if the life-span of the storage used by routine 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 routine B may result in the reference by B to the (nonexistent) object A, producing unpredictable (and invariably incorrect) results.

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

Determining Library Entrypoints

Each library EPF contains a list of its entrypoints. This entrypoint list names the routines in the library that may be called from outside the library. You use BIND to create the entrypoint list when you build your library.

By default, BIND puts the names of all routines in the library into the library's entrypoint list. You can use the ENTRYNAME subcommand to exclude some routines from the library's entrypoint list. However, a program cannot directly call a subroutine unless the subroutine is included in the library's entrypoint list.

Unlisted routines can be called by other routines in the same library. Another routine in the same library can provide a calling program with the address of an unlisted routine. In this way, some routines in a library can be made externally available while others are reserved for internal use by other routines in the same library.

Entryname Conventions

Prime reserves the following entrynames for Prime-supplied subroutines:

- All entrynames containing a $ sign (for example: GV$GET, CL$PIX, ECL$CC, SLEEP$)
- All entrynames listed in Table 5–2

Do not use any of these entrynames unless you specifically plan to replace a Prime supplied subroutine with one of your own.

Note The IX-mode C compiler automatically adds the G$ prefix to all external references unless they are declared as having `fortran` storage class (a Prime extension to the C language). For example, a call to the `printf` routine creates a reference to G$PRINTF. This forces IX-mode programs to call the IX-mode versions of subroutines. If you create your own IX-mode C library, you must prefix the entrynames with G$. For example, if you call a routine in your library using the C statement

```
y=my_sum(x);
```

the library entryname for the called routine must be G$MY_SUM.

Table 5–2. Subroutine Entrynames Reserved by Prime

ACKRCT	ENCRYP	LIBTBL	QPOST	T1IB
ADD_QREC	EPF_ERR	LIST_SRL	QUITHD	T1IN
ADOCRD	EPF_RL	LNGCMP	QUOTE_	T1OB
ADQREC	ERASE	LOCK	R0BASE	T1OU
ADRESS	ERROPN	LOG_EVEN	R3FALT	TBLRED
AD_CMD	ERRRTN	LOG_RECO	RDASC	TIDEC
ALLOC	ERRSET	MCSDAT	RDBIN	TIHEX
APPEND	EVAL_A	MCSTOD	RDNPAG	TIMDAT
APROTO	EXIT	MOVB	RDPRCN	TIMREC
ATLIST	EXTRAC	MOVE	RECYCL	TIMSLT
ATMAIN	FILERR	MOVEB	RELGRP	TIOCT
ATTDEV	FILHER	MOVWDS	REMANS	TM3270
AVAIL	FINDPG	MSGCTL	REMUSR	TMDISP
BCKUPB	FIND_U	NETCHK	REPOST	TNOU
BSCMAN	FIND_UID	NETFIG	RESTART_	TNOUA
C1IN	FNDREC	NETPRC	RGSTRY	TODEC
CALFC_	FINDWRD	NETSET	RJDBG	TOHEX
CFI	FORCEW	NEWS	RJMNIT	TONL
CIRLOG	FREE_DES	NOTDST	RJPROC	TOOCT
CKINST	FREE_O_R	NPXPRC	RMLOCK	TRNRCV
CKNDNM	FSCHOC	NXTLIN	RPTSPL	TRTYPE
CLEARS	GALLKS	OAUSER	RQUEST	TRVERSIO
CLREAD	GCHAR	OERRTN	RSTBL	TSATRC
CLRLIN	GETADR	OPNDFL	SAL_HP	TSTAMP
CLR_FLDS	GETENT	OPNDOC	SCANB	UDTDRY
CLSDOC	GETERR	OPNQFL	SCHAR	UDTF07
CMD_POST	GETNPG	OWL2	SEARCH_C	UNCAN_ME
CMD_PRE_	GETNYB	P1IB	SEARCH_H	UNLOCK

Table 5–2. Subroutine Entrynames Reserved by Prime (continued)

CNIN	GETREG	P1IN	SECBLD	UNPACK_A
COMANL	GETSLT	P1OB	SEGCON	UPCASE
CONTRL	GET_REPL	P1OU	SELANG	UPDATE_S
CRAWL_	GFILKS	P2UPCS	SETATT	USERID
CREUFD	GINFO	PACK_BIT	SETNAM	USRPRM
DATE_A	GORDNC	PACK_CHA	SETREG	VMMSG
DCTEXS	GOREAD	PACK_INT	SET_SRL	VMMSG2
DECR_HOP	GTDOCR	PARS_ATT	SET_VERS	VMMSG3
DEFILE	GTWORD	PARTCL	SFR_CFSC	VREMID
DELAY	GUSLKS	PASSWD	SFR_HP	WHATIT
DELAY_	HASH_U	PCREAT	SHRLIB	WRASC
DELETE	HASH_UID	PEXIT	SH_CMD	WRBIN
DELETE_Q	ICMTB_	PFIL2A	SLAVE	WRITLINE
DELOAS	ICPL_	PFLM9E	SLAVER	WRTPG
DH3270	ICS2CT	PHDBG	SOUR3	XLACPT
DIDNUM	ID	PINIT	SPLCHK	XLASGN
DIRSER	INCPTR	PINLNK	SRWREC	XLCLR
DISPLA	INITP1	PK2LDV	STK_EX	XLCLRA
DMIDAS	INIT_NPX	PRIBLD	STPNC	XLCONN
DMLCP	INIT_O_S	PRICON	STRBL	XLGOON
DNUMID	INTCM_	PRVSB_	STRTPH	XLGVVC
DOSSUB	IQNET	PTRAP	STUFF	XLUASN
DPTINI	IQUSER	PUTBL	SUBMIT	XMTRCV
DPTOFF	DPTOFF	ISFEPF	PUTSLT	SWFBK_
DRAIN_QU	JUSTRT	PUT_HOP	SWFIM_	
EM3270	LCKGRP	QPARSE	SWINTQ	

Usually you use the names by which routines were defined in the source code as entrynames in the library's entrypoint list. If you want to use different names, you can have BIND change the names of routines, but this requires an extra step in the build process. To keep matters simple, choose the names carefully at coding time and stick with them.

Building a Library EPF With BIND

Once you have decided on the entrypoint list and class of your library, you can build it with BIND. The following examples show you typical build sequences for program-class and process-class dynamic library EPFs. Even if you are planning to register your library, you should first build and test it as a dynamic EPF. Once the library is functioning successfully as a dynamic EPF it is simple to rebuild it as a registered EPF. Registered EPFs are described in Chapter 6.

For a program-class library EPF, the build sequence is

```
OK, BIND library-EPF-filename
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LIBMODE -PROGRAM
Library is program class.
: LOAD module-1   Load the compiled code of your library.
: LOAD module-2
       .
       .
       .
: ENTRYNAME name-1 [name-2 ...] Create entrypoint list.
: LIBRARY library-name         Load any special libraries
                                required.
       .
       .
       .
: LIBRARY     Load the standard system library if needed.
: FILE
OK,
```

The important points to note are

- You may provide a filename for the library EPF either on the BIND command line or with the FILE subcommand at the end of the build sequence. If you fail to supply a filename, BIND gives the library EPF the same name as the first module loaded.
- The LIBMODE –PROGRAM subcommand defines the EPF as a program-class library.

- The compiled code of your subroutines can be in one file or in several separately compiled modules. You may find it easier to maintain the library if you code and compile the routines separately. When you load several modules, you can either specify several LOAD subcommands, as in the example, or name all the modules on the same subcommand line.
- You can name all the entrypoints on the same ENTRYNAME subcommand line, as in the example, or give a separate ENTRYNAME subcommand for each entrypoint.
- You may find it useful for debugging purposes to issue the MAP subcommand before you file the EPF. If you want the locations of common areas to appear on the map, issue the RESOLVE_DEFERRED_COMMON subcommand before the MAP subcommand.

For a process-class library EPF, the build sequence is

```
OK, BIND library-EPF-filename
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LIBMODE -PROCESS
Library is process class.
: LOAD module-1
: LOAD module-2
        .
        .
        .
: ENTRYNAME name-1 [name-2 ...]
: LIBRARY library-name
        .
        .
        .
: LIBRARY PROCESS_CLASS    Load the process class special
                           library.
: LIBRARY
: FILE
OK,
```

The differences between the process-class and program-class build sequences are

- The LIBMODE –PROCESS command declares the EPF to be a process-class library.
- The subcommand LIBRARY PROCESS_CLASS loads a special binary library used by process-class library EPFs. This library contains routines that cause all dynamic allocation performed by your library routines to be done in process-class memory rather than program-class memory. For more information, see the section Storage Allocation Issues below.

Automatically Generating Entrynames: You can have BIND automatically generate a list of entrynames by issuing the ENTRYNAME -ALL subcommand. Before loading the modules containing your subroutines, issue ENTRYNAME -ALL. After loading the modules containing the subroutines you want listed in the entrypoint list, issue ENTRYNAME -NONE. This tells BIND not to add entrynames for any subroutines subsequently loaded.

The sequence of subcommands is

```
: ENTRYNAME -ALL
: LOAD module-1          These modules contain routines
: LOAD module-2          that you want listed as entrypoints.
     .
     .
     .
: ENTRYNAME -NONE
: LOAD other-module-1    These modules contain routines
: LOAD other-module-2     that you don't want listed as
                         entrypoints.
     .
     .
     .
: LIBRARY [special-library-1 ...]     If needed
```

Always issue the ENTRYNAME -NONE subcommand before loading any libraries with 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. The ENTRYNAME subcommand is further described in the *Programmer's Guide to BIND and EPFs.*

Changing Entrynames: An entryname is normally the same as the name of the subroutine in your compiled code. If you want a subroutine's entryname to be different from the name under which the subroutine was declared in the source code, use the CHANGE_SYMBOL_NAME subcommand.

To change the name of an entryname, issue the CHANGE_SYMBOL_NAME subcommand, then issue the ENTRYNAME subcommand. For example, suppose you define a subroutine called FOO, using the FORTRAN 77 statement

```
SUBROUTINE FOO(X)
```

If you want to declare this subroutine as an entrypoint called BAR in your library EPF, give the following subcommands to BIND:

```
: CHANGE_SYMBOL_NAME FOO BAR
: ENTRYNAME BAR
```

If you have BIND generate the entryname list with the ENTRYNAME –ALL subcommand, give the CHANGE_SYMBOL_NAME subcommand after you load the subroutine modules.

The CHANGE_SYMBOL_NAME subcommand is further described in the *Programmer's Guide to BIND and EPFs.*

Creating Binary Libraries With EDIT_BINARY

You can greatly increase the convenience of using your library EPF by creating a binary library with dynts to the library EPF entrypoints. After successfully building your library EPF using BIND, use EDIT_BINARY to generate a corresponding binary library. You use this binary library when you build other EPFs that call routines in the library EPF.

Without such a binary library, when you build an EPF that calls routines in your library EPF, you need to specify individual dynts for each called routine (using the DYNT subcommand of BIND). With such a binary library, you can build an EPF that calls routines in your library EPF exactly as you build EPFs that call Prime-supplied subroutines. Simply load your binary library during the BIND session, just as you load Prime-supplied binary libraries.

EDIT_BINARY is further described in Chapter 9.

Installing Libraries

In order to use a library EPF and its corresponding binary library, you must carry out three operations:

- Install the binary library in an appropriate directory.
- Install the library EPF in an appropriate directory.
- Modify your ENTRY$ search rules.

These operations can be performed in any sequence.

Installing Binary Libraries

Where you install your binary library depends on who you want to have access to it.

You can install your binary library in the top-level directory LIB. LIB is a directory of binary libraries. It is accessible to all users.

You can install your binary library in a directory accessible to a specific set of users.

If you place your library in the LIB top-level directory, users can load it during a BIND session using the subcommand

LIBRARY *library_name*

If you place your library in another directory, users must load it using the full pathname with either

LIBRARY *library_pathname*

or

LOAD *library_pathname*

Note The only difference between the LIBRARY and LOAD subcommands in BIND is that LIBRARY adds LIB> to *library–name* when you don't specify a pathname for the library. The command

LIBRARY *library_name*

is equivalent to the command

LOAD LIB>*library_name*

The advantage of installing your binary library in the LIB top-level directory is that users can build with it using the same subcommand format they use to build with Prime-supplied libraries. Installing your binary library in another directory is useful when you want to restrict use of the binary library to a specific group of users.

Installing Library EPFs

PRIMOS reserves the top-level directory LIBRARIES* for library EPFs. You can install your library EPF either in LIBRARIES*, or in some other directory accessible to a specific group of users.

Wherever you install it, a library EPF is only accessible to the dynamic linking mechanism if the library's pathname appears in an ENTRY$ search list. Even if you install a library EPF in the LIBRARIES* top-level directory, a user cannot run programs that call the library unless the library's pathname is included in the user's ENTRY$ search rules.

Setting ENTRY$ Search Rules

The dynamic linking mechanism uses the ENTRY$ search list to resolve dynamic links during program execution. An ENTRY$ search list contains the pathnames of libraries to be searched for routines. Any library EPF that you construct must be listed in the ENTRY$ search list of all users that will run programs that call routines in your library EPF. This section describes the steps needed to include a library EPF in an ENTRY$ search list.

The dynamic linking mechanism can use either of the following search lists:

- The systemwide default ENTRY$ search list, SYSTEM>ENTRY$.SR
- A private ENTRY$ search list maintained by each user

Private search lists usually include the system default search list along with additional library pathnames. Therefore programs run by users with private search lists can normally make use of both the system default and user-specific libraries.

Modifying the System Default ENTRY$ Search List

You can make your library EPF accessible to users throughout the system by adding the library's pathname to SYSTEM>ENTRY$.SR. Typically, you request that the System Administrator install your library EPF in the LIBRARIES* directory, and add its pathname to the system default ENTRY$ search list.

Caution Typically, you do not have access to SYSTEM>ENTRY$.SR unless you are the System Administrator. If you modify it, it is possible to accidentally render it unusable, such as by inserting a duplicate search rule. If ENTRY$ is corrupted, not only will users be affected, but a subsequent cold start of the system may render the supervisor terminal nearly ineffective. In such a situation, you will be unable to use ED or EMACS to fix the file, since both editors themselves reference faulted IPs to call system subroutines via the dynamic linking mechanism.

The solution to this problem is to use the non-shared editor, NSED, to fix the default search list file. NSED runs under PRIMOS II, and therefore does not ever reference faulted IPs. Rebooting the system would then load the corrected default search list file.

When a user logs in, PRIMOS supplies the user with the current system default search lists. A change to the system default search lists only affects a user when the user's process is reinitialized. Users who are already logged in when the System Administrator modifies a system default search list need to issue the SET_SEARCH_RULES command (with the –DEFAULT option) or the INITIALIZE_COMMAND_ENVIRONMENT (ICE) command to update their system default search lists.

Creating and Modifying Private Search Lists

Alternatively, you can create or modify private ENTRY$ search lists for users who run programs that call your library EPF. Usually such private search lists include both the system default ENTRY$ search rules and the pathnames of user-specific libraries. You include the system default ENTRY$ search rules by including the –SYSTEM search rule in the private search list.

For example, suppose you want to make your library EPF available only to a restricted group of users who have access to the directory MY_GROUP>LIBRARIES. If you install MY_LIBRARY.RUN in the directory MY_GROUP>LIBRARIES, you can create the following ENTRY$ search list for group members:

```
–SYSTEM
MY_GROUP>LIBRARIES>MYLIBRARY.RUN
```

Suppose you save this file as MY_GROUP>ENTRY$.SR. Each group member who plans to run a program that accesses your library EPF must then issue the command

```
SET_SEARCH_RULES MY_GROUP>ENTRY$.SR
```

This sets the user's ENTRY$ search list.

When a user logs in or reinitializes the user process (by issuing the ICE command), all search lists are automatically reset to system defaults. For this reason, users who want to always use a private search list should add the SET_SEARCH_RULES command to their LOGIN.CPL or LOGIN.COMI file. This insures that the user is always accessing the proper search list.

Note 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 ENTRY$ search list that includes your library EPF. Ask the user to issue the LIST_SEARCH_RULES command (abbreviated LSR) and be sure that the pathname of your library EPF is listed.

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.

The SET_SEARCH_RULES, LIST_SEARCH_RULES, and LLENT commands are described in the *PRIMOS Commands Reference Guide*; the search rules facility is described in the *Advanced Programmer's Guide II: File System.*

Registered EPFs

6

Registered EPFs provide an efficient means to implement shared programs and libraries. How much a given program can benefit from registration depends on how the program is coded. This chapter describes

- Deciding whether a program is a good candidate for registration
- Writing programs for registered EPFs
- Building registered EPFs with BIND
- Creating binary libraries that reference registered library EPFs
- Registering EPFs (performed by the System Administrator)
- Accessing registered EPFs
- Getting information about registered EPFs

Should You Register an EPF?

You can register both program EPFs and library EPFs. Registering an EPF offers a number of advantages both for system and individual user performance:

- Registered EPFs share linkage, reducing the system working set.
- Dynamic links in shared linkage are pre-snapped, reducing execution time.
- Per-user data is generally initialized faster, reducing startup time.

Nearly all EPFs can take advantage of these features, but some EPFs will benefit more than others. This section gives you some guidelines for identifying EPFs that are likely to benefit most.

Shared Linkage

One of the major advantages of registered EPFs is that they can share linkage. Shared linkage allows registered EPFs to use less system resources and to startup and run faster.

- Because only one copy of the shared linkage needs to be maintained on the system, a registered EPF being run by several users occupies fewer system resources than a dynamic EPF version of the same EPF. Systemwide, the registered EPF version uses less memory and requires less paging per user to file system disks.
- Dynts in shared linkage are snapped at registration time. Since these dynts need not be snapped at runtime, the registered EPF runs faster.

The more linkage an EPF has, the more it can benefit from registration. Highly modular programs tend to have more linkage. Each routine in your program or library that is not shortcalled has an Entry Control Block (ECB). Registration places these ECBs into shared linkage. Calls to subroutines generate IPs. Registration places many of these IPs in shared linkage.

You can share linkage for internal calls to routines within the program EPF. You can also share linkage for external calls to routines in registered library EPFs and PRIMOS direct entries. You cannot share linkage for external calls to dynamic library EPFs or static-mode libraries. Therefore, if your program calls mostly PRIMOS direct entries, internal routines, and external routines in other registered EPFs, then your program or library can benefit from registration.

Other Factors

Even programs that do not generate a large amount of shareable linkage may benefit from registration. PRIMOS creates and stores an initialized copy of per-user data and linkage at registration time. When a user invokes a registered EPF, this copy can be quickly mapped to the user's address space so the EPF starts up faster. In a dynamic EPF, per-user linkage/data segments must be expanded from templates each time a user invokes the EPF.

Although nearly all programs can benefit in some way from registration, keep in mind that a registered EPF continues to occupy system resources until it is unregistered. A registered EPF remains mapped to shared segments, and PRIMOS must store information about it even if no one invokes it. Frequently used and widely used programs and routines are therefore better candidates for registration than programs or routines rarely used or run by only a few users. In general, linkage-intensive programs tend to benefit the most from registration.

Registered EPFs perform fewer paging operations to the user's file system disk space, but may require more paging disk space than dynamic EPFs. This should improve I/O performance, although the total number of page faults may not decrease. It may be necessary to increase the size of the paging disk to support large registered EPFs.

Figure 6-1 illustrates how registered EPFs have the ability to use both shared and per-user linkage. In most cases, shared is preferred.

Dynamic Program EPF • Single entrypoint • Direct invocation • Per-user linkage	Dynamic Library EPF • Multiple entrypoints • Indirect invocation • Per-user linkage
Registered Program EPF • Single entrypoint • Direct invocation • Shared linkage (optional per-user linkage)	Registered Library EPF • Single entrypoint • Direct invocation • Shared linkage (optional per-user linkage)

Figure 6-1. Comparison of Properties Between Dynamic EPFs and Registered EPFs

Creating Registered EPFs

Prime recommends that you build registered EPFs in the following stages:

1. Build, test, and debug a program as a dynamic EPF.
2. Relink with BIND to create a registrable EPF.
3. Test the registrable version by running it unregistered.
4. Have the System Administrator register the EPF.
5. Add the -PUBLIC rule to the appropriate search lists.
6. Test the registered version.

These stages apply to the creation of both registered program EPFs and registered library EPFs. By following this sequence, you can quickly isolate problems that occur at the coding, building, or registration stages.

Compiler Support

For a program EPF or a library EPF to benefit from registration, it must have been compiled with a compiler that supports Z frame organization. Traditionally, Prime compilers divided code into three frames: the procedure frame, the data frame, and the stack frame. The data frame contained data, common areas, and linkage information. More recent Prime compilers support a fourth frame, the Z frame. The Z frame is a separate frame that holds the linkage information that was previously stored in the data frame.

As of Translator Family Revision T3.0, all Prime-supplied compilers, except FTN, support Z frame organization. The Z frame is automatically generated during all program compiles. These compilers generate code that can take advantage of all the benefits of EPF registration. Z frame organization should have no effect on the linking, loading, or execution of programs that are not registered. Therefore, recompilation of programs is never required, and is only recommended if you intend to use that program to build a registered EPF.

Table 6-1 shows the first release of each compiler that fully supports registered EPFs.

Table 6-1. Compiler Support for Registered EPFs

Compiler	*First Release Supporting Registered EPFs*
PMA	PRIMOS Rev. 21.0
C	T1.2
CBL	T2.0
COBOL85	Compiler Rev. 1.0
FTN	Not supported
F77	PRIMOS Rev 20.2
Pascal	T3.0
PLIG	T3.0
PL/I	T3.0
VRPG	PRIMOS Rev. 20.2

Code compiled with the FTN compiler or with older versions of the other compilers can be used to build a registered EPF, but performance benefits are smaller because linkage cannot be shared. For full registered EPF support, you should convert FTN code to F77 and recompile. Code compiled with earlier versions of other compilers should be recompiled with more recent compiler versions, as indicated in Table 6–1.

Existing PMA programs require some minor source code changes to create registrable EPFs with shared linkage. Appendix A shows you how to create registrable EPFs with PMA.

Coding Guidelines

For programs written in high-level languages, no programming restrictions apply. The only guideline is to follow good programming practices. In particular, you need not worry that a highly modular program will start up or run slowly because it uses a lot of linkage. Because shared dynts are snapped and linkage segments created when you register the EPF, registered EPFs start quickly even when they contain a great deal of linkage.

Remember that shared dynts are only used for calls to PRIMOS direct entries and registered library EPFs. PRIMOS snaps shared dynts to these libraries when you register the EPF. Your program EPF may also contain per-user (non-shared) dynts. All calls to routines in dynamic library EPFs and shared static-mode libraries are per-user dynts. You can also specify per-user dynts to routines in registered library EPFs. Per-user dynts are snapped during program execution.

If your EPF calls routines in user-written or other non-Prime supplied libraries, linkage to those routines can only be shared if those libraries are registered. To get maximum benefit from registration of a program EPF, you should also register the library EPFs that contain frequently called routines.

Note If you maintain shared static-mode programs or shared static-mode libraries on your system, you should consider converting them to registered EPFs. Appendix C provides guidelines for converting static-mode code to registered EPFs.

To maximize performance of registered EPFs, you should keep in mind that when you invoke a registered EPF, PRIMOS initializes all of the registered library EPFs linked to your registered EPF by shared dynts. Therefore, a program that calls routines in many registered EPF libraries may be less efficient than one that calls routines in fewer registered EPF libraries. To improve initialization efficiency of registered EPFs

- Consolidate frequently called routines into fewer registered library EPFs.
- Reduce the size of these registered library EPFs by removing rarely called, personal, and obsolete routines. Registered EPFs should contain code that is frequently called by multiple users, not rarely used routines or programs used by only a single user.
- Establish per-user (rather than shared) dynts to rarely called routines in registered EPF libraries. (You establish establish the dynt type when you use BIND to build the EPF.)
- Avoid nesting registered library EPFs with shared dynts. An example of this is a registered EPF that calls a routine in a registered library EPF and that routine calls a routine in another registered library EPF. In this case, if all of these links are shared dynts, PRIMOS initializes all three registered EPFs (the program EPF and the two library EPFs) even if these routines are never called during program execution. (Nesting shared dynts also complicates EPF registration, as described later in this chapter.)

PRIMOS direct entries do not require initialization. You should specify PRIMOS direct entries as shared dynts, regardless of how frequently they are called.

Compiler Options

EPFs must be compiled in V-mode or I-mode.

The -PBECB compiler option should be avoided, in most cases, because this option substantially reduces the amount of shared code. Chapter 4 provides further details on the -PBECB option.

Building With BIND

It is recommended that you initially build and test all programs as dynamic EPFs, then rebuild the tested version as a registered EPF. Once you have built and tested an EPF as a dynamic EPF, it is not difficult to rebuild it with BIND to create a registered EPF.

BIND provides several subcommands for building a registered EPF.

Using the -REGISTER Option

The only requirement for building a registered EPF is to add the -REGISTER subcommand option to the first subcommand of the build sequence:

To build a registered program EPF, use the subcommand

PROGMODE -REGISTER

To build a registered library EPF, use the subcommand

$$\textbf{LIBMODE} \left\{ \begin{array}{l} \textbf{- PROGRAM} \\ \textbf{- PROCESS} \end{array} \right\} \textbf{- REGISTER}$$

When you specify -REGISTER to create a registered library EPF, you must also specify either -PROGRAM or -PROCESS for the per-user linkage/data of the registered library EPF. Chapter 5 describes how to determine whether a library EPF should be program-class or process-class.

Setting the Dynt Type

When using BIND to build an EPF, you specify whether the dynts are shared or per-user (non-shared).

- Dynts to dynamic library EPFs and shared static-mode libraries must be per-user.
- Dynts to PRIMOS direct entries can be either shared or per-user. Normally, you want them to be shared in order to get the full benefits of registration. Shared dynts are preferable for all references to PRIMOS direct entries, because PRIMOS direct entries do not require initialization.
- Dynts to registered library EPFs can be either shared or per-user. A shared dynt saves dynt-snapping time during execution, but requires PRIMOS to initialize the registered library EPF at the beginning of program execution. PRIMOS initializes the registered library EPF even if the dynt to that library is never executed. Therefore, it is advantageous to establish this dynt as a shared dynt if you expect the program to use the dynt during normal execution. If, however, the dynt is almost never used (for example, a call to an error handler), it may be advantageous to establish the dynt as a per-user dynt.

You set the dynt type by using the DEFAULT_DYNT_TYPE subcommand and the DYNT subcommand.

In most cases, you should set the default dynt type to -SHARED when using BIND to build a registered EPF. You set the default dynt type with the subcommand

DEFAULT_DYNT_TYPE -SHARED

Give this subcommand before loading any binary libraries. This assures that BIND places all dynts optimally in your registered EPF.

You can respecify the DEFAULT_DYNT_TYPE several times during a BIND session.

You can use the DYNT subcommand to declare the dynt type of individual dynts. If you do not specify a dynt type with the DYNT subcommand, BIND first defaults to the most recent setting of DEFAULT_DYNT_TYPE; if you have not specified a default dynt type during that BIND session, BIND defaults to per-user.

Table 6-2 shows the dynt types that result from all possible combinations of DEFAULT_DYNT_TYPE and DYNT subcommand values:

Table 6-2. Dynt Types Established by BIND

	DEFAULT_DYNT_TYPE		
DYNT	***-SHARED***	***-PER_USER***	***DEFAULT_DYNT_TYPE Not Issued***
-SHARED	Shared	Shared	Shared
-PER_USER	Per-user	Per-user	Per-user
-DEFAULT or no option specified	Shared	Per-user	Per-user

Building With the DYNT Subcommand: You can place dynts to user-supplied routines in a registered EPF runfile by using the DYNT subcommand of BIND. When you do this, be sure that BIND puts the dynts in the appropriate place:

- Dynts to routines in registered library EPFs and PRIMOS direct entries should be shared.
- All other dynts should be per-user.

The DYNT subcommand defaults to -DEFAULT, so that BIND places the dynt according to the most recent setting of the DEFAULT_DYNT_TYPE subcommand. If you have not specified a default dynt type during the BIND session, the DYNT subcommand without options creates a per-user dynt.

The following example shows how to use the DYNT -SHARED subcommand to create shared dynts to three subroutines in a user-supplied registered library EPF:

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: PROGMODE -REGISTER
EPF is registrable program type.
: LO PROG_A
: LI
: DYNT -SHARED FOO, MOO, BAR
BIND COMPLETE
```

You can do the same thing by setting the default dynt type to shared:

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: PROGMODE -REGISTER
EPF is registrable program type.
: LO PROG_A
: LI
: DEFAULT_DYNT_TYPE -SHARED
: DYNT FOO, MOO, BAR
BIND COMPLETE
```

If you have a user-supplied library EPF that you link to using the DYNT subcommand of BIND, you may want to consider creating a binary library containing dynts to the entrypoints. You can then link to multiple routines by loading a single binary library instead of creating individual dynts to each routine. EDIT_BINARY greatly simplifies the process of creating binary libraries. You establish dynt types when using EDIT_BINARY to build a binary library. The next section, Dynt Types and Binary Libraries, gives guidelines for building binary libraries.

Dynt Types and Binary Libraries

Each dynt in a binary library can be of dynt type SHARED, PER_USER, or DEFAULT. BIND cannot change a SHARED or PER_USER dynt type specified in a binary library. BIND can change a DEFAULT dynt type specified in a binary library. Dynt type DEFAULT means that BIND must set the dynt type to either PER_USER or SHARED when you submit the binary library to BIND. Setting the dynt type during a BIND session is described in the previous section.

Binary Libraries Supplied by Prime: In binary libraries supplied by Prime, dynts to PRIMOS direct entries are marked as SHARED, and dynts to routines in registered library EPFs are marked as DEFAULT. Dynts to other libraries are marked as PER_USER. You specify the dynt type for the dynts marked DEFAULT when you load this binary library during a BIND session. BIND sets DEFAULT dynts to the dynt type specified by the BIND DEFAULT_DYNT_TYPE command. For example, if you set the BIND default dynt type to SHARED before loading one of these binary libraries, BIND places all dynts marked SHARED or DEFAULT in shared linkage and place dynts marked PER_USER in per-user linkage.

User-created Binary Libraries: When you use EDIT_BINARY to create a binary library, you should mark dynts to PRIMOS direct entries as SHARED and dynts to routines in registered library EPFs as DEFAULT. Mark dynts to routines in other libraries as PER_USER. This give the user the flexibility to assign either dynt type to dynts for PRIMOS direct entries and routines in registered library EPFs. In most cases, you would assign dynt type SHARED to these dynts during the BIND session.

All dynts in pre-Rev. 23.0 binary libraries are considered per-user dynts. Even if dynts in these binary libraries reference routines now in registered library EPFs, BIND still puts these dynts in per-user linkage. If you build with your own pre-Rev. 23.0 binary libraries, you may want to rebuild these binary libraries in order to generate more efficient registered EPFs. The section, Setting Dynt Types in Binary Libraries, shows you how.

Loading a Binary Library: You can place dynts in a registered EPF by loading a binary library. You load a binary library using either the LOAD or LIBRARY subcommand of BIND. The following example shows a build sequence with a user-created binary library. It creates a registered EPF named TEST, using a binary library named TEST_LIB.

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: PROGMODE -REGISTER
EPF is registrable program type.
: LO TEST
: DEFAULT_DYNT_TYPE -SHARED
: LI MYLIBS>TEST_LIB
: LI
BIND COMPLETE
: FILE
OK,
```

Because the appropriate dynt types (DEFAULT or PER_USER) were set when you used EDIT_BINARY to build the binary library, you can just set the DEFAULT_DYNT_TYPE during the BIND session to resolve the dynts of dynt type DEFAULT.

This BIND build sequence uses a pathname with the LIBRARY subcommand, because the user-supplied library is not installed in the LIB top-level directory. If the library was in LIB, you would use the LOAD subcommand rather than the LIBRARY subcommand.

The next section shows you how to use EDIT_BINARY to build binary libraries that conform to Prime standards.

Setting Dynt Types in Binary Libraries

To build binary libraries according to Prime standards, you should

- Mark dynts to registered library EPFs and PRIMOS direct entries as -DEFAULT.
- Mark dynts to routines not in registered library EPFs or PRIMOS direct entries as -PER_USER.

Note Do not mark dynts to routines in dynamic library EPFs as DEFAULT. If these dynts are marked as default, then the standard BIND build sequence given above puts them in shared linkage. An EPF with such dynts in shared linkage cannot be registered successfully.

Using the Prime standard for binary libraries provides maximum flexibility and simplifies program maintenance. Other options are available, although not recommended: When using EDIT_BINARY to build a binary library, you can mark dynts to PRIMOS entrypoints and registered library EPFs as PER_USER, but registered EPFs built with such a binary library do not gain the benefits of dynt sharing. You can also use EDIT_BINARY to mark these dynts as SHARED. In this case, you need not set the DEFAULT_DYNT_TYPE subcommand to SHARED when you use BIND to build with such a binary library. However, you cannot use this binary library to create per-user dynts.

Using the EDIT_BINARY DYNT Subcommand: You can create dynts of all types in a binary library using the DYNT and DEFAULT_DYNT_TYPE subcommands of EDIT_BINARY. These EDIT_BINARY subcommands are very similar to the corresponding BIND subcommands. The principal difference is that during an EDIT_BINARY session you can establish a dynt of dynt type DEFAULT.

A dynt type of DEFAULT is resolved when you submit the binary library to a BIND session. Table 6-3 shows the possible combinations of

DEFAULT_DYNT_TYPE and DYNT subcommands during an EDIT_BINARY session:

Table 6-3. Dynt Types Set by the DYNT Subcommand of EDIT_BINARY

	DEFAULT_DYNT_TYPE			
DYNT	***-SHARED***	***-PER_USER***	***-DEFAULT***	***DEFAULT_DYNT_TYPE not issued***
-SHARED	Shared	Shared	Shared	Shared
-PER_USER	Per-user	Per-user	Per-user	Per-user
-DEFAULT ***or option not specified***	Shared	Per-user	Default, determined at BIND time	Per-user

Chapter 9 contains a complete reference to the EDIT_BINARY subcommands.

Using the EDIT_BINARY READ Subcommand: If you create dynts using the READ subcommand with no options, EDIT_BINARY automatically sets the dynt types according to the Prime standard:

- When reading a registrable library EPF, dynts are marked as DEFAULT.
- When reading a dynamic library EPF, dynts are marked as PER_USER.

You can use the READ subcommand's -PER_USER, -SHARED, and -DEFAULT options to override this automatic dynt type setting. (Note that -DEFAULT is not the same as supplying no options. -DEFAULT creates default dynts. With no options the dynt type depends on the type of library being read.)

Table 6-4 shows the possible dynt types created by the READ subcommand during an EDIT_BINARY session.

Table 6-4. Dynt Types Set by the READ Subcommand of EDIT_BINARY

Options	*Values*
-SHARED	Shared
-PER_USER	Per-user
-DEFAULT	Default, determined at BIND time

No option specified	Set by library type:
	Registered = Default
	Dynamic = Per-user

The EDIT_BINARY DEFAULT_DYNT_TYPE subcommand is only used with the DYNT subcommand; it does not affect dynt types established using READ. Chapter 9 contains a complete reference to the EDIT_BINARY subcommands.

Two EDIT_BINARY Examples: The following example creates a binary library containing default dynts by reading a user-created registered library EPF:

```
OK, EDIT_BINARY
[EDIT_BINARY Rev. T3.0-23.0 (c) Prime Computer,Inc. 1990]
: RFL
: READ MY_LIB.RUN
  Library creation date:        90-12-03.14:06:24.Mon
  Library type:                 registered program
  Number of entries:            4
End processing MY_LIB.RUN.
: SFL
: FILE MY_BIN
OK,
```

You can check the dynt types created with the LIST_CONTENTS subcommand.

```
OK, EDIT_BINARY
[EDIT_BINARY Rev. T3.0-23.0 (c) Prime Computer,Inc. 1990]
: OPEN MY_BIN
: LIST_CONTENTS -DYNTS
Contents of file "MY_BIN":
INITRT (DF)               BAR (DF)       MOO (DF)
FOO (DF)
End of list.
: QUIT
OK,
```

Here the display shows the dynt types as default (DF).

The next example shows how to use the READ subcommand to specify a dynt type that differs from the standard type for that library. For example, to create per-user dynts when reading a registered library EPF, you would give the -PER_USER option with the READ command.

```
OK, EDIT_BINARY
[EDIT_BINARY Rev. T3.0-23.0 (c) Prime Computer,Inc. 1990]
: RFL
: READ MY_LIB -PER_USER
  Library creation date:        90-12-03.14:06:24.Mon
  Library type:                 registered program
  Number of entries:            4
```

```
End processing MY_LIB.
: SFL
: FILE MY_BIN
OK, EDIT_BINARY
[EDIT_BINARY Rev. T3.0-23.0 (c) Prime Computer,Inc. 1990]
: OPEN MY_BIN
: LIST_CONTENTS -DYNTS
Contents of file "MY_BIN":
INITRT (PU)                BAR (PU)        MOO (PU)
FOO (PU)
End of list.
: QUIT
OK,
```

Rebuilding Old Binary Libraries

Prime recommends that you use EDIT_BINARY to rebuild your old binary libraries, marking all dynts to routines not in registered library EPFs as -PER_USER. If you also use BIND to rebuild your dynamic library EPFs as registered library EPFs, you can mark dynts to routines in these libraries as -DEFAULT when you rebuild the binary libraries.

Supplying Initialization Routines

BIND provides the INIT_ENTRY subcommand, which allows you to specify a routine in a registered EPF that PRIMOS will automatically execute when you register the EPF. INIT_ENTRY is only meaningful if building a registered EPF.

Typically, you establish such a routine to initialize data areas at registration time. In a dynamic EPF, PRIMOS recreates the data image in memory from the dynamic EPF file each time a user invokes the program. It uses information in the EPF file to initialize the data image. In a registered EPF, PRIMOS does not access the EPF file when a user invokes the program; instead, PRIMOS copies the data image from shared memory into the user's memory space. Therefore, it may be necessary to initialize data areas at registration time, when these data areas are stored in shared memory.

The INIT_ENTRY subcommand allows you to include a data initialization routine when you build a registered EPF. When the System Administrator registers the EPF, PRIMOS executes the initialization routine, then saves the data image with the appropriate initialization. A routine specified by INIT_ENTRY can accept arguments supplied at registration time, so you can use such a routine to carry out installation-specific initialization.

To specify a registration-time initialization routine, use the subcommand

INIT_ENTRY *entryname*

during the BIND session. *entryname* must specify an existing routine (with an Entry Control Block (ECB)). If it does not, or if the EPF you are building is not a registered EPF, BIND displays an error message.

For example, suppose you are building a registered library EPF called LIB_B that includes an initialization routine called INIT_DB. You can have PRIMOS execute INIT_DB when LIB_B is registered by giving the INIT_ENTRY subcommand during the BIND build session:

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) Prime Computer,Inc. 1990]
: LIBMODE -PROCESS -REGISTER
Library is registrable process class.
: ENTRYNAME -ALL
All successive entrypoints will be added to the
 entrypoint table
: LO LIB_B
: LI
BIND COMPLETE
: INIT_ENTRY INIT_DB
Init Program ECB is INIT_DB at -0004/000122
: FILE
OK,
```

The System Administrator can pass parameters to the initialization routine at registration time by using the -INIT option of the REGISTER_EPF command. See the section Registering an EPF, below, for details.

Note When you create an initialization routine in a registered EPF, remember that your program may have dependencies: a set of library EPFs that must be registered in order to register your EPF. These library EPFs can also contain initialization routines. In fact, you can have more than one of these library EPF routines perform initialization. If you do this, request that the System Administrator use the -INIT_DEPENDENCY_LIST option of the REGISTER_EPF command to specify the order in which initialization routines should be executed. For more details on dependency lists and REGISTER_EPF options, see the section Registering EPFs, below.

Creating Shared Common Areas

Registered EPFs give you more flexibility than dynamic EPFs in creating common areas. Dynamic EPFs can share read-only common areas. Registered EPFs can share both read/write and read-only common areas.

The ability to create shared read/write common areas makes it possible to use registered EPFs to share data among both programs and processes.

Using the -SHARE option of the ALLOCATE subcommand of BIND, you can specify that a common area be placed in the shared portion of a registered EPF. You can use the -SHARE option to share both read/write and read-only common areas.

With dynamic EPFs, only read-only data areas can be shared, and the sharing is done implicitly by BIND without programmer intervention.

For a detailed description of this procedure, see Chapter 7, Shared Data.

Testing an EPF

You should test an EPF prior to registering it. To test a complex EPF, the following steps are recommended:

1. Use BIND to build the EPF as a dynamic EPF.
2. Test the dynamic EPF.
3. Use BIND to rebuild the EPF as a registrable EPF.
4. Test the registrable EPF.
5. Register the registrable EPF.
6. Test the registered EPF.

When testing an EPF, you should be aware of how the different versions of an EPF use shared segments. See Table 6-5.

Table 6-5. Memory Assignment for EPF Versions

EPF Version	*Per-user Segments*	*Shared Segments*
Dynamic	Data and linkage	Procedure code and literals
Registrable	Data, linkage, procedure code and literals	
Registered	Data	Procedure code, literals, and linkage

Executing a registrable EPF requires more resources than running the same program as either a dynamic EPF or a registered EPF. In general, registrable EPFs should only be used for testing purposes.

Registering EPFs

Only a System Administrator can register an EPF. An EPF is considered **registered** when the System Administrator uses the REGISTER_EPF command to perform the registration process. PRIMOS may suspend the registration of an EPF until the successful registration of other related EPFs. A suspended EPF cannot be executed as a registered EPF; however, the dynamic EPF or registrable EPF version of the EPF may be executed.

Although only a System Administrator can register an EPF, as a programmer you should understand the registration process so you can provide EPFs that can be registered successfully. You may also need to provide information and support to the person who registers your programs. You may want to provide a command file or CPL program that contains the necessary registration commands to your System Administrator. This section gives you the basic information you need.

The System Administrator can register an EPF using either the REGISTER_EPF command or the EPF$REG subroutine. Registration is usually performed as part of the cold-start procedure, but the System Administrator can register an EPF at any time. The REGISTER_EPF command is described in the *Operator's Guide to System Commands*. The EPF$REG subroutine is described in the *Subroutines Reference II: File System*.

Dependency Lists

The basic rule of EPF registration is that shared dynts in a registered EPF may only link to

- Routines in other registered EPFs
- PRIMOS direct entries

The reason for this restriction is straightforward: in order to share a dynt, PRIMOS must reference the routine in a fixed location with a virtual address that is the same for all users. In other words, the called routine must also reside in shared memory. PRIMOS direct entries and routines in registered library EPFs meet this condition. Routines in dynamic library EPFs do not.

Note A registered EPF can, of course, call routines in dynamic library EPFs. Links to routines in dynamic library EPFs are per-user dynts, not shared dynts. A registered EPF can also call routines in other registered EPFs using per-user dynts. Libraries linked to your registered EPF by per-user dynts are not registration dependencies.

Direct and Indirect Dependencies: Successful registration of an EPF with shared dynts to routines in other EPFs depends on registration of the called EPFs. Registered EPFs called by shared dynts in your registered EPF are said to be **dependencies** of your registered EPF. Such dependencies may be both direct and indirect:

- A **direct dependency** is a registered EPF called by a shared dynt in your registered EPF.
- An **indirect dependency** is a registered EPF not directly called by your EPF, but called by a dependency of your EPF.

For example, suppose the registered EPF PROG_A uses shared dynts to call routines in the libraries LIB_B and LIB_C and that one of the called routines in LIB_C uses a shared dynt to call a routine in LIB_D which in turn calls a routine in LIB_E. In this case, LIB_B and LIB_C are direct dependencies for PROG_A. LIB_D and LIB_E are indirect dependencies.

In order to register an EPF successfully, all direct and indirect dependencies must also be registered. When you build a registered EPF with shared dynts to other registered EPFs, you must be aware of all direct and indirect dependencies. If any of these dependencies are not already registered on your system, they must also be registered in order to register your EPF successfully.

Usually, registered library EPFs supplied by Prime are registered at system cold start. You can use the LIST_REGISTERED_EPF command to determine which libraries are actually registered on your system. If your EPF has any dependencies that are not registered, you need to supply this information to your System Administrator so that all these required libraries can also be registered.

Registered EPF States

The registration process is designed so that you can register an EPF and its dependencies in any order. PRIMOS allows you to register an EPF even if all of its dependencies have not been registered first. Such a registered EPF is placed in a suspended state and cannot be executed until all dependencies have been resolved. (If this were not the case, you would need to register the EPFs in order, beginning with indirect dependencies, and you would be unable to register EPFs with circular dependencies.)

During registration, each EPF passes through two phases, each phase having two states:

1. Linkage phase
   - o Linkage uninitialized
   - o Linkage initialized
2. Invocation phase
   - o Invocation suspended
   - o Invocation ready

Linkage: In the first phase of the registration process, PRIMOS locates and resolves (snaps) all of the registered EPF's shared dynamic links.

A registered EPF is **uninitialized** when any shared dynamic links remain unsnapped.

A registered EPF is **initialized** when all shared dynamic links have been snapped.

One cause of uninitialized linkage is a shared dynt that references a routine in a dynamic library EPF or shared static-mode library. Only per-user (non-shared) dynts can access routines in these libraries. To correct this problem, you must either use BIND to rebuild your program EPF, specifying the per-user dynt type for these references, or place the referenced routines in a registered EPF.

As PRIMOS resolves shared dynamic links, it adds the names of the library EPFs referenced by these dynamic links to a list of dependencies. After attempting to resolve all shared dynamic links, PRIMOS proceeds to the invocation phase and checks the contents of this dependencies list.

Invocation: In the second phase of the registration process, PRIMOS determines if all of the dependencies of the registered EPF have been registered.

A registered EPF is **suspended** when it is registered, but the EPF cannot be executed. A registered EPF is suspended under any of the following conditions:

- The registered EPF was not successfully initialized.
- An EPF that is a direct or indirect dependency cannot be located.
- An EPF that is a direct or indirect dependency has not been initialized.
- An EPF that is a direct or indirect dependency is suspended because one of its dependencies cannot be located or has not been initialized.

If a registered EPF was not successfully initialized, its status is uninitialized and suspended. If a registered EPF was successfully initialized, but has some problem with its dependencies, its status is initialized but suspended.

A registered EPF is **ready** when all the shared dynamic links are resolved and all direct and indirect dependencies are ready as well.

Cross-checking: PRIMOS maintains a list of all suspended EPFs. Whenever a new EPF is registered, PRIMOS automatically cross-checks this list of suspended EPFs.

- When you begin registering an EPF, PRIMOS checks all of its suspended EPFs to see if your EPF can be used to resolve any previously unresolved dynamic links.
- When an EPF is updated from suspended to ready status, PRIMOS checks to see if it can update the status of any EPFs that call the newly updated EPF.

Because PRIMOS constantly updates the status of all affected EPFs, you can register an EPF and its dependencies in any order. When you register an EPF before registering all of its dependencies, the EPF is marked as suspended. Once you have registered all of the dependencies, the EPF's state is updated to ready.

Multiple EPF Registrations

The System Administrator can register more than one EPF with the same name. This permits you to supply a new version of a registered EPF without requiring that the System Administrator unregister the old version. This makes it possible to update a registered EPF without corrupting some user's executing environment.

Note For simplicity, it is recommended that, whenever possible, the System Administrator register all registered EPF at the same time during coldstart, and that you avoid re-registering or unregistering EPFs while there are active users on the system.

As each EPF is registered, it is given a registration number. The first version of each EPF has registration number 1. Subsequent versions have higher numbers. The LIST_REGISTERED_EPF command displays the registration number of each registered EPF.

The following rules govern the use of multiple registered EPFs:

- Executing a program EPF automatically executes the highest-numbered version of that program EPF.

- An executing program EPF continues execution, unaffected by the registering of a new version of that EPF or its dependencies.
- A registered EPF accesses the dependencies that were the highest-numbered versions at the time that the EPF was registered. You must, therefore, re-register a program EPF to access a new version of a library EPF.

Unregistering EPFs

The System Administrator can use the UNREGISTER_EPF command or the EPF$UREG subroutine to remove a registered EPF from the registered EPF database. If an unregistered EPF is on the dependency list of some other registered EPF, then the invocation status of the other EPF is changed to suspended. For example, if EPF_A depends on EPF_B, then unregistering EPF_B causes EPF_A's invocation status to be changed to suspended.

By default, the UNREGISTER_EPF command only unregisters EPFs that are not currently in use. However, the System Administrator can specify the -FORCE option to unregister an EPF that is currently in use by some user or that is a dependency of an EPF that is currently in use. If this happens, the user's executing environment is corrupted, and the executing program will probably fail.

Setting Paging Disk Space

After registering an EPF, the System Administrator may need to change the size of the system's paging disk. PRIMOS copies the per-user linkage and data of a registered EPF from DTAR1 shared memory segments into each user's DTAR3 memory segments when the user invokes the registered EPF. This operation uses the system paging disk. The actual paging disk requirements depend both on the size of the registered EPF and the anticipated number of concurrent users of that EPF.

To calculate the paging disk requirements for a registered EPF, you need the map output from BIND. A typical Segment portion of the BIND map looks like this:

```
Segment Type                Low    High   Top
 -0004  DATA              000000 000700 000700
 -0002  SHARED PROC       001000 001074 001076
 +0000  PROC              177777 000000 001000 EMPTY
```

To calculate the paging disk requirements you first need to determine the size of each segment, in records. To do this, you take the Top offset for each negative segment and place it in the formula: (Top + 17777 / 20000) * 10. (All numbers are octal.) Using the BIND map example, you would calculate:

-0004 : ((700 + 17777) / 20000) * 10 = (20677/20000) * 10 = 1*10 = 10
-0002 : ((1076 + 17777) / 20000) * 10 = (21075/20000) * 10 = 1*10 = 10

Therefore, rounded to the nearest whole number, the size of each segment is octal 10, or decimal 8 records.

To determine the total paging space requirement, you must add the size of the SHARED PROC segment (in this case, decimal 8 records) to the total non-shared requirement. To get the total non-shared requirement, multiply the size of the non-shared DATA segment (in this case, decimal 8 records) by the anticipated number of concurrent users sharing the registered EPF, plus 1 user. Concurrent users are those who have the EPF mapped in their user memory; after execution, an EPF remains mapped in the user's memory either until the user reinitializes the process (by logging out, for example) or until the user explicitly unmaps the EPF. If we assume 30 concurrent users, the calculation (in decimal numbers) for this example is

8 DATA segment records * (30 + 1 users) = 248 non-shared records

248 non-shared records + 8 SHARED PROC segment records = 256 total records.

In this example, therefore, the extra paging requirement for running this registered EPF is 256 records.

Using Registered EPFs

To use a registered EPF, you may need to first check that the EPF has, in fact, been successfully registered (using the LIST_REGISTERED_EPF command) and check your user search lists (using the LIST_SEARCH_RULES command) to make sure they can access the registered EPF. Once these tasks are completed, you can invoke the registered EPF.

There are no restrictions on the mixing of EPF types. A registered program EPF can call routines in registered library EPFs and dynamic library EPFs. A dynamic program EPF can call routines in registered library EPFs and dynamic library EPFs. You can maintain otherwise identical programs and libraries of both types on the system. However, EPF registration is most advantageous when a registered program EPF calls routines in registered library EPFs or PRIMOS direct entries.

Getting Information on Registered EPFs

You can use three PRIMOS commands to check the status of registered EPFs. These commands are available to all users.

LIST_EPF -REG Command: Use the -REG option of the LIST_EPF (LE) command to get a listing of all registered EPFs on your system. The display shows the invocation state of each registered EPF as either `Suspended` or `Ready`. LIST_EPF -REG displays information about EPFs in the system's registered EPF database rather than registered EPFs mapped to any given user's address space. LIST_EPF is documented in the *PRIMOS Commands Reference Guide*.

```
OK, LIST_EPF -REG

2 Process-Class Library EPFs.

(registered) (Ready)      BOOTLEG.RUN
(registered) (Ready)      BOOTLEG.RUN

2 Program-Class Library EPFs.

(registered) (Ready)      PRIMIX_IX_CC_LIBRARY.RUN
(registered) (Ready)      PRIMIX_IX_LIBCURSES.RUN

1 Program EPF.

(registered) (Suspended) CALL_LIB.RUN

OK,
```

In this example, LIST_EPF -REG shows that CALL_LIB.RUN, a registered EPF, is in the suspended state. This could be due to an unresolved shared dynamic link or a suspended dependency. To determine why this registered EPF is suspended, you can use the LIST_REGISTERED_EPF command.

LIST_REGISTERED_EPF Command: Use the LIST_REGISTERED_EPF (LRE) command to get detailed information on the status of a specific registered EPF on your system. This command lists the dependencies of a registered EPF, and indicates whether each dependency is Direct or Indirect. This command also lists the names of any unresolved entrypoints. This list of unresolved entrypoints lists all unresolved shared dynamic links; it does not list unresolved per-user dynamic links. LIST_REGISTERED_EPF is documented in the *Operator's Guide to System Commands.*

```
OK, LIST_REGISTERED_EPF

2 Process-Class Library EPFs.
===========================

BOOTLEG.RUN   (Ready)       (Registration #  2)

  No resolved/specified dependencies found.

  No unresolved entrypoints.

BOOTLEG.RUN   (Ready)       (Registration #  1)

  No resolved/specified dependencies found.

  No unresolved entrypoints.

2 Program-Class Library EPFs.
===========================

PRIMIX_IX_CC_LIBRARY.RUN   (Ready)    (Registration #  1)

  No resolved/specified dependencies found.

  No unresolved entrypoints.

PRIMIX_IX_LIBCURSES.RUN   (Ready)     (Registration #  1)

  No resolved/specified dependencies found.

  No unresolved entrypoints.

1 Program EPF.
===========================

CALL_LIB.RUN   (Suspended)   (Registration #  1)

  No resolved/specified dependencies found.

  Unresolved Entrypoint List:
        BAR
        FOO
        MOO
OK,
```

In this example, LIST_REGISTERED_EPF shows that three references to external routines (BAR, FOO, and MOO) could not be resolved during registration. This could happen because the library EPFs that contain these entrypoints have not yet been registered, or it could result from an error in specifying the names of the dynts when BIND was used to build the registered EPF. You can use the LIST_LIBRARY_ENTRIES command to determine which library EPF contains these entrypoints.

LIST_LIBRARY_ENTRIES -REG Command: Use the -REG option of the LIST_LIBRARY_ENTRIES (LLENT) command to display a list of entrypoints contained in registered library EPFs on your system.

```
OK, LLENT BOOTLEG.RUN -REG
(ring 3 epf) BOOTLEG.RUN
Ring3 Proc-Class Lib EPF,
1 Total Entrypoints,
1 Selected Entrypoints
QED$E
```

LIST_LIBRARY_ENTRIES is documented in the *PRIMOS Commands Reference Guide.*

EPF$ISREADY Subroutine: Use the EPF$ISREADY subroutine to return the status of a specific registered EPF to a user program. EPF$ISREADY returns a value of 1 if the registered EPF is ready and a value of 0 if the registered EPF is suspended. EPF$ISREADY is documented in the *Subroutines Reference II: File System.*

Setting Search Rules for Registered EPFs

In order to execute registered EPFs, you must include the -PUBLIC search rule in your user search lists.

- To execute registered program EPFs, a user's COMMAND$ search list must include the -PUBLIC rule. This tells PRIMOS to search the registered EPF database for command names.
- To execute an EPF that dynamically links to registered library EPFs, a user's ENTRY$ search list must include the -PUBLIC search rule. This tells PRIMOS to search the registered library EPFs for entrynames during dynamic linking.

The -PUBLIC search rule is included in the default system search rules for COMMAND$ and ENTRY$. If you use the default search rules, no change is necessary.

PRIMOS does not search registered EPFs in a specific order. This presents a problem if two registered library EPFs contain a routine with the same name. The simplest solution is to make sure that all registered EPF routines have unique names. This, however, is not always possible. When duplicate names are necessary, you can use -PUBLIC search rules to specify the names of specific registered EPF in each user's search list, as follows:

```
-PUBLIC registered_epf_name1
-PUBLIC registered_epf_name2
-PUBLIC
```

In this example, PRIMOS first searches *registered_EPF_name1*, then searches *registered_epf_name2*, then searches all other registered EPFs. You should end a list of -PUBLIC search rules with a -PUBLIC search rule with no EPF name option. This tells PRIMOS to search all the registered EPFs.

Note In most cases, you should put in your search list both a -PUBLIC search rule to access an EPF as a registered EPF and a standard search rule to access the EPF as a dynamic EPF. This is especially critical for core system libraries. If you do not include a search rule for the dynamic EPF version, unregistering an EPF library makes that library inaccessible to all programs.

To make sure that the registered version of an EPF is executed in preference to the dynamic version, be sure that the -PUBLIC search rule precedes any search rules that lead to dynamic versions of registered EPFs. For example, suppose you have placed a dynamic EPF version of the library LIB_A.RUN in a directory named MY_LIBS. If you later register LIB_A.RUN, you can be sure that programs link to the registered version by placing the -PUBLIC search rule before the MY_LIBS > LIB_A.RUN search rule in your ENTRY$ search list.

To list the contents of your search lists, use the LIST_SEARCH_RULES command. To modify your search lists, use the SET_SEARCH_RULES command. Both commands are documented in the *PRIMOS Commands Reference Guide*. You can use the MONITOR_SEARCH_RULES command to monitor ENTRY$ search list performance. This command is documented in the *PRIMOS User's Release Document*. For further details on search rules, refer to the *Advanced Programmer's Guide II: File System*.

Registered EPF Access

Unlike dynamic EPFs, registered EPFs cannot be ACL protected. This is because registered EPFs do not reside in the file system. You should be aware that all users have access to registered EPFs on your system.

- All users can display the names of the registered EPFs on your system using the LIST_REGISTERED_EPF and LIST_EPF -REG commands.
- All users can execute registered program EPFs.
- All users can link to the entrypoints of registered library EPFs.

If you want to restrict access to a registered EPF, you should code the EPF so that it must be invoked by a dynamic EPF interlude. You can then restrict access by setting ACL protection on the interlude.

Invoking Registered Program EPFs

Registered Program EPFs can be executed either as commands or command functions. The -PUBLIC search rule must be included in your COMMAND$ search list in order for PRIMOS to find and execute registered program EPFs. If the -PUBLIC search rule is not included or the EPF is not registered, PRIMOS returns the `Not found` error message.

In addition to the registered EPF, you may also have other copies of the same program EPF on your system: a dynamic program EPF version (used for testing), and a registrable EPF version (the program EPF you submitted to the System Administrator for registration). Both of these versions execute successfully, but neither takes advantage of EPF registration. A registrable EPF executed in this way is mapped and executed entirely in per-user segments. Because PRIMOS is unable to share any of the registrable EPF, it executes inefficiently. Therefore, care should be taken not to execute one of these versions rather than the actual registered EPF.

If you supply the RESUME command with the complete pathname of the EPF, PRIMOS does not use the COMMAND$ search list, and therefore does not execute the registered EPF. It instead executes the pathname version.

If you incorrectly place the -PUBLIC search rule in the COMMAND$ search list, PRIMOS may find and execute another version of the program EPF, rather than the registered EPF. The -PUBLIC search rule must be closer to the beginning of the search list than the name of any directory that contains a version of the EPF with the same name.

You can call a registered program EPF from another program using the CP$ subroutine. The EPF$RUN subroutine and the other EPF$ subroutines (EPF$INVK, EPF$MAP, etc.) cannot be used with registered EPFs. Refer to the *Advanced Programmer's Guide III: Command Environment* for further details.

7 Shared Data

EPFs provide three ways to share data:

- You can share data among programs within a process using a process-class library EPF.
- You can share data among processes using a shared read/write common area in a registered EPF.
- You can access shared data areas in static memory.

This chapter shows you how to use these data sharing methods.

Using Process-class Library EPFs

Process-class library EPFs provide a way to share data among programs within a single user process. The technique is simple:

- Allocate the shared data structure as a static data area in a process-class library EPF.
- Have one or more subroutines handle access to the data area.

Since the static data in the linkage/data area of a process-class library EPF is only initialized once per user process, all programs that access the data through the library subroutines have access to the same data.

In effect, this technique provides automatic dynamic allocation of per-process shared data areas:

- PRIMOS allocates space for the shared data when it allocates the linkage/data area of the library EPF.
- PRIMOS initializes the shared data area once per process when it initializes the library EPF.

The most straightforward technique is to have a library entrypoint to handle each data manipulation operation: writing to the shared data area, reading the data, reinitializing the data, and the like. Programs that access the shared data then do so by calling these entrypoints. The sample program TUBES_LIB.C below shows you how to create a library EPF that handles shared data in this way.

Note that your subroutines must provide for the possibility that more than one program attempts to update the shared data simultaneously. The section Providing for Simultaneous Updates, below, describes some techniques for doing this. The sample program TUBES_LIB.C includes provisions for simultaneous updates.

You must also observe one important restriction when creating process-class libraries to do this kind of per-process data sharing: because of the restriction on library class mixing, the library routines that access the shared data must not call routines in program-class libraries. Remember that language-directed I/O is handled by program-class subroutines, so your routines should not do any language-directed I/O. For further details on these subjects, refer to Chapter 5.

Note As of Rev. 23.0, PRIMOS does not provide explicit means to allocate and link to a common area dynamically. The technique given here implicitly provides much of the same functionality for per-process shared data. The subroutine or subroutines that access the data area act as a gateway for EPFs that need to use the data.

Using Shared Read/Write Common Areas

Registered EPFs give you the ability to create shared read/write common areas. Since shared areas are not reinitialized for each user, they can be used to share data among several user processes.

The technique is much like the one used to share data among programs using a process-class library EPF, but in this case, you can use either a library EPF or a program EPF.

1. Write a routine or routines that manipulate data in a common area data structure.
2. Use BIND to build a registered EPF with these routines. Use the –SHARE and –ACCESS options of the ALLOCATE subcommand to define the data structure as a shared read/write common area. The section Creating a Shared Common Area With BIND, below, shows you how to do this.

The simplest approach is to create a registered library EPF much like the process-class dynamic library EPF described in the previous section. You could, for example, write one routine to update the data structure and another to return its contents. You can test such a library as a process-classdynamic library EPF that shares data among programs and then rebuild it as a registered library EPF to share data among processes. The program examples below show a library built and tested in this way.

If you create a registered program EPF, you can write an interface that allows users to read and write to the common area. However, you may find a program EPF more difficult to test and debug than a library EPF. A more flexible technique would be to put the user interface in a program EPF and the shared data manipulation routines in a registered library EPF.

Again, you must write code that deals with simultaneous attempts to update the common area. See the section Providing for Simultaneous Updates, below, for information on this topic.

Creating Shared Common Areas With BIND

You can have BIND allocate a shared read/write common data area using the –SHARE and –ACCESS options of the ALLOCATE subcommand. The format is

ALLOCATE *symbol_name size* **–SHARE –ACCESS READ/WRITE**

The arguments are

symbol_name	The name by which the shared data structure is referenced in the EPF.
size	A decimal number that gives the size of the data area in 16-bit halfwords.
–SHARE	Tells BIND to place the data area in a shared segment.
–ACCESS READ/WRITE	Sets read/write access to the shared area.

You must give the ALLOCATE subcommand before you load any routines that reference the common data area.

Debugging an EPF that shares data in this way can be difficult, since you cannot create a shared read/write common area in a dynamic EPF. If you are creating a library EPF, one possibility is to create a process-class dynamic library EPF first, as described in the previous section. You can test the library's ability to share data among programs within a single process. You can then rebuild the library as a registered EPF, allocating the shared data structure as a shared read/write common area. You can then test its ability to share data among processes.

Using Static Shared Data

Allocating static segments for shared data is inconsistent with the dynamic approach encouraged by EPFs and is not recommended when other techniques are available. The SYMBOL subcommand of BIND does, however, provide you with the means to build an EPF that addresses statically allocated data.

The technique is as follows:

1. Allocate the appropriate static memory for your data object.
2. Initialize the data area either at system coldstart or at user login.
3. When using BIND to build an EPF that addresses the shared data area, use the SYMBOL subcommand to specify the address of the data area.

Allocating Space

You allocate space for the shared data object in static segments. For data that is to be shared among programs within a process, you allocate the segments from per-user static memory (segment numbers between 4000 and the first dynamic segment). Remember that PRIMOS does not manage access to this memory for you. You must take care to see that your data area does not conflict with memory used by other programs.

For data that is to be shared among processes, you use shared static segments. Consult with the System Administrator to find out which segments are available for sharing. The System Administrator must then make the shared segments available at system coldstart using the SHARE command.

Initializing the Data

You must explicitly initialize the shared data area because PRIMOS does not initialize static segments when initializing an EPF.

For data in per-user static segments, you must be sure that each user's data area is initialized before its first use during a terminal session. The simplest way to do this is to write an initialization program that writes the correct initial values to the shared data area. Have each user's LOGIN.CPL invoke this initialization program so that it is run once at each login or command environment initialization.

For data in shared static segments, perform the initialization at cold start when the segments are shared. You can do this in two ways:

- Run an initialization program at cold start. The System Administrator or PRIMOS.COMI should invoke the initialization program immediately after the SHARE command.

- Create a static-mode memory image with SEG and load it with the SHARE command. Consult the *SEG and LOAD Reference Guide* for information about how to do this.

Using the SYMBOL Subcommand

When you use BIND to build an EPF that references the shared data area, you must use the SYMBOL subcommand to specify the area's address. The format is

SYMBOL *symbol_name definition* [*size*]

The arguments are

symbol_name	The name by which the data area is referenced in your program
definition	The address of the data area in *segment_number/offset* format
size	An optional decimal number that give the size of the data area in 16-bit halfwords

After you give the SYMBOL subcommand, all references in your program to *symbol_name* refer to the data area.

When you use static segments to share data you should be aware of two important restrictions:

- You are responsible for managing and initializing the static memory. Other programs may overwrite your data area.
- You must use BIND to rebuild your EPF any time the shared data area address is changed. You must provide the new address with the SYMBOL subcommand. This makes EPFs that address static segments less easy to maintain than EPFs that do not.

Because of these restrictions, you should avoid using static segments to share data among EPFs if possible.

Also remember that any code that updates a shared data area must be able to handle concurrent updates. The following section discusses this issue.

Providing for Concurrent Updates

Whenever more than one program or process can reference shared data concurrently, you must consider the possibility that one access may interfere with another. If you use shared data structures that you have allocated in EPFs or static segments, your code must be designed to handle concurrent attempts to access the data.

In the case of multiple processes, the problem can occur during process exchange. If one process is interrupted while accessing the shared data, another process may also access the shared data while the first is waiting to resume execution. When you have either multiple processes or multiple programs within a single process, a similar problem can occur when one program is suspended (because you typed CONTROL–P, for example). If you suspend a program while it is accessing shared data, the data may be accessed by another program while the first is suspended. If you later restart the first program, the shared data may now be corrupted.

It can be difficult to spot potential concurrency problems in source code. Individual program statements are often interruptible. For example, the statement

```
item = item + 1
```

might well generate code that could be interrupted between reading the value of *item* and incrementing its value. If such an interruption occurs, and another program subsequently alters the value of *item*, the first program will still write the incremented value of the old *item* when it resumes execution. This may not be what your software expects. Such problems can be a difficult to diagnose, since they may occur only irregularly and may be very difficult to reproduce.

Writing code that avoids these problems is a topic beyond the scope of this book. This section indicates some of the resources available in PRIMOS for dealing with concurrent updates and suggests some solutions.

Atomic Update Routines in PMA

You can write PMA subroutines that update individual shared locations atomically. Such routines are coded so that they can check the validity of the shared location and update it in a way that is immune to interruption. The following PMA routines show how to use the STAC instruction to atomically update the contents of memory locations holding 16-bit integer data.

The first routine, COND_STORE.PMA, atomically attempts to replace the old contents of a 16-bit memory location with a new value. The function return value indicates whether the atomic update was successful. The routine is called with three halfword integers, and returns as its function value a halfword integer. In PL/I, 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 SPLIT_LINKAGE

* The SPLIT_LINKAGE keyword allows the subroutine to
* share linkage when it is built as a registered EPF

          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 second routine, INCREMENT.PMA, atomically increments the value of a 16-bit memory location by 1 and returns the new value. The PL/I calling sequence is as follows:

DCL INCREMENT ENTRY (FIXED BIN(15))
RETURNS (FIXED BIN(15));

new_value = INCREMENT (***old_value***);

The PMA source is as follows:

```
          SEG SPLIT_LINKAGE
          RLIT
          SYML
*
          SUBR INCREMENT,ECB
*
          LINK
ECB      ECB INCREMENT,,VARIABLE,1
          PROC
*
INCREMENT EQU *
          ARGT
*
TRY_AGAIN LDA VARIABLE,*
          TAB
          A1A
          STAC VARIABLE,*
          BCNE TRY_AGAIN
*
          PRTN
*
          DYNM VARIABLE(3)
*
          END
```

You can change the INCREMENT routine into a DECREMENT routine by changing the A1A instruction to S1A.

By themselves, these routines only update a single 16-bit memory location at a time. You can rewrite the routines to update 32-bit locations, but you can also use the 16-bit routines to protect arbitrarily large data areas by having the routines set access flags on the data. The TUBES_LIB.C example shows you how to use the COND_STORE routine in this way.

Other Techniques

You may be able to use other PRIMOS facilities to protect shared data areas from simultaneous access or even to avoid using shared memory altogether.

Synchronizers and Semaphores: You may be able to use event synchronizers or semaphores to coordinate access to shared data resources. For detailed information on semaphores, see *Subroutines Reference III: Operating System.* For detailed information on event synchronizers, see *Subroutines Reference V: Event Synchronization.*

Using the File System and Global Variables: You can avoid many data sharing problems by using the file system or global variables. You can use the file system to share data of all types both among programs and among processes. You can use global variables to share small amounts of character data among programs within a process. In both cases, PRIMOS handles allocation and updating for you. The *Advanced Programmer's Guide II: File System* includes a detailed discussion of interprocess communication via the file system.

InterServer Communication Facility: ISC provides a means of passing both large and small amounts of string data between two processes. Programming with ISC requires calls to many subroutines, but relieves you of the necessity of managing shared static memory and concurrent updates. If your programs need to pass blocks of string data, consider using ISC. You can find a complete description of ISC in *Subroutines Reference V: Event Synchronization.*

A Data Sharing Example

The following C language program, TUBES_LIB.C, is a simple demonstration of the EPF data sharing techniques discussed in this chapter. When built as a registered EPF library, TUBES_LIB provides routines for passing character data between processes. Data are passed through a set of shared data structures called tubes. The array of tubes is allocated as a shared read/write common area when the program is built as a registered EPF. Library entrypoints provide a way for programs to open, close, read from, and write to tubes.

```
/* tubes_lib.c is a library of routines that can be used to manage tubes. */
/* Tubes are shared buffers that can be used to pass character data       */
/* between user processes. tubes_lib.c provides an orderly way for a      */
/* process to acquire the number of a free tube and to write to or read   */
/* from it. The following entrypoints are available:                      */
/*                                                                        */
/* open_tube     Returns the number of a free tube                        */
/* close_tube    Marks a specified tube as free                           */
/* write_to_tube Writes one character to a specified tube                 */
/* read_tube     Returns one character from a specified tube              */
/*                                                                        */
/* Note that the library does not provide any means to control access     */
/* to tubes, to prevent processes from reading and writing to closed      */
/* tubes, or to prevent processes from closing tubes still in use. The    */
/* calling processes are responsible for preventing such conflicts.       */

#include "stdio.h"

#define NTUBES 16    /* number of tubes allocated on system */
#define TSIZE  128   /* length of tube buffer */
#define FREE 0       /* tube not currently in use */
#define USED 1       /* tube in use */
#define NONEFREE -1 /* no tubes currently available */
#define OK 1
```

```
#define FAILED 0

/* THE ARRAY OF TUBE STRUCTURES */
/* link as shared read-write */

struct tube_blk{
    short in_use;     /* equals USED if tube is in use, else FREE */
    short c_count;    /* number of chars in tube buffer */
    char *bot_ptr;    /* first element in tube buffer */
    char *top_ptr;    /* last element */
    char *read_ptr;   /* read from here */
    char *write_ptr;  /* write to here */
    char tube[TSIZE]; /*tube buffer itself */
} tube_table[NTUBES];

/* ROUTINE TO INITIALIZE THE SHARED ARRAY  */
/* this is run at registration time as an init_entry */

int init_tubes(){

    int count;

    for (count = 0; count < NTUBES ; count++) close_tube(count);
    return(OK);
}

/* CALL TO CLOSE A TUBE */
/* closes a tube whether it is open or not */

int close_tube(tube_num)
int  tube_num;
{
    extern struct tube_blk tube_table[];

    tube_table[tube_num].c_count = 0; /* tube is empty */
    tube_table[tube_num].bot_ptr = tube_table[tube_num].tube;
    tube_table[tube_num].top_ptr = &(tube_table[tube_num].tube[TSIZE-1]);
    tube_table[tube_num].read_ptr = tube_table[tube_num].top_ptr;
    tube_table[tube_num].write_ptr = tube_table[tube_num].top_ptr;
    tube_table[tube_num].in_use = FREE; /* not in use, do this last */
    return(OK);
}

/* CALL TO OPEN A TUBE  */
/* returns either the tube number or NONEFREE if no tubes are available */

int open_tube(){

    /* updates the in_use flag conditionally to prevent more than    */
    /* one process grabbing the same tube.                           */
    fortran short constore(); /* the atomic update routine */
    extern struct tube_blk tube_table[];

    int tube_num; /* tubes numbered from 0 to PNUM -1 */

    tube_num = 0; /* try first tube */
    while (tube_num < NTUBES){
         if (tube_table[tube_num].in_use == FREE){ /* currently free */
              if (constore(tube_table[tube_num].in_use,USED,FREE)){
                   return(tube_num); /* return the tube number if atomic */
              }                        /* update was successful  */
         }
         tube_num++;                   /* else try another tube */
    }
    return(NONEFREE);                  /* no free tubes this time through */
}

/* CALL TO WRITE A CHARACTER TO A TUBE */
/* updates the character count atomically to allow  */
/* simultaneous reads and writes                    */

int write_to_tube(c, tube_num)
char c;
int tube_num;
{
    extern struct tube_blk tube_table[];
    fortran short increment(); /* atomic increment routine */

    while(tube_table[tube_num].c_count >= TSIZE); /* wait until there's room */

    /* or we could return a value that indicates a full tube and let the */
    /* calling application decide whether to wait */

    *(tube_table[tube_num].write_ptr) = c;
    (tube_table[tube_num].write_ptr)++; /* move pointer and wrap if needed */
    if (tube_table[tube_num].write_ptr > tube_table[tube_num].top_ptr)
       tube_table[tube_num].write_ptr = tube_table[tube_num].bot_ptr;
    increment(tube_table[tube_num].c_count); /* atomically update char count */
 }

/* CALL TO READ A CHARACTER FROM A TUBE */
/* updates the character count atomically to allow  */
/* simultaneous reads and writes                    */

char read_tube(tube_num)
int tube_num;
{
    extern struct tube_blk tube_table[];
    fortran short decrement();  /* atomic decrement routine */
    char c;
```

```
   while(tube_table[tube_num].c_count == 0); /* wait for chars to read */
/* or we could return a value to indicate that there are no characters */
/* to read and let the calling program decide whether to wait.        */

   c = *(tube_table[tube_num].read_ptr);
   (tube_table[tube_num].read_ptr)++;
                    /* increment pointer and wrap if needed */
   if (tube_table[tube_num].read_ptr > tube_table[tube_num].top_ptr)
      tube_table[tube_num].read_ptr = tube_table[tube_num].bot_ptr;
   decrement(tube_table[tube_num].c_count);
                   /* atomic decrement of char_count */
   return c;
}
```

Note the use of atomic updates by TUBES_LIB.C:

- The tube structure uses the short integer *variable in_use* to indicate whether a given tube has already been selected for use. The *open_tube* subroutine uses the COND_STORE routine to update this variable atomically when searching for a tube that is not in use.
- Reads and writes use the INCREMENT and DECREMENT subroutines to update *c_count* atomically. This prevents the generation of invalid values if *read_tube* and *write_tube* are interrupted.

The *in_use* variable is a simple example of how a flag can protect a data structure from simultaneous access. As long as each process that writes to a tube first calls *open_tube* and uses the tube number returned, simultaneous writes cannot occur.

The simultaneous access protection implemented in TUBES_LIB.C is the minimum required by cooperating processes to maintain the integrity of the shared data structures. If processes fail to cooperate, the contents of the tube structures can become corrupted. If processes don't call *open_tubes*, for example, there is no means to prevent them from reading or writing closed tubes, writing simultaneously to the same tube, and so on. Note also that there is nothing to prevent a tube from being closed while a write or read is occurring. Reading and writing processes therefore need to establish a clear protocol for terminating communication before closing a tube.

Additional checks can be added to the library to relieve the calling processes of these responsibilities. Ultimately, however, the security of shared read/write data structures like tubes depends on cooperation by user processes. Because shared read/write data areas reside in shared segments, user programs can always get access to them and intentionally or unintentionally change their contents.

Building TUBES_LIB: The recommended procedure for building and testing a registered library EPF that shares data is to build it first as a process-class dynamic library EPF. The dynamic library EPF can then be tested by calling it from several programs within a single process. After such testing the library can be rebuilt as a registered EPF and subjected to multiple process testing. The BIND sequence to build TUBES_LIB as a process-class dynamic library EPF is

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LIBMODE -PROCESS
Library is process-class.
: ENTRYNAME -ALL
All successive entrypoints will be added to the
entrypoint table
: LO TUBES_LIB
: ENTRYNAME -NONE
Successive entrypoints will not be added to the
entrypoint table
: LO COND_STORE
: LO INCREMENT
: LO DECREMENT
BIND COMPLETE
: FILE
OK,
```

After testing the dynamic version, you can then rebuild the library as a registered EPF.

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LIBMODE -PROCESS -REGISTER
Library is registerable process-class.
: ALLOCATE TUBE_TABLE 2000 -SHARE -ACCESS READ/WRITE
New symbol PIPE_TABLE is 2000(003720) words long
The area is shared and its access type is read/write.
: ENTRYNAME -ALL
All successive entrypoints will be added to the
entrypoint table
: LO TUBES_LIB
 "TUBE_TABLE" :smaller redefinition of common.
: ENTRYNAME -NONE
Successive entrypoints will not be added to the
entrypoint table
: LO COND_STORE
: LO INCREMENT
: LO DECREMENT
BIND COMPLETE
: INIT_ENTRY INIT_TUBES
Init Program ECB is INIT_TUBES at -0004/004130
: FILE
OK,
```

Note the important differences in the build sequence:

- The –REGISTER option declares the library as registrable.
- The ALLOCATE subcommand is used to reserve shared read/write space for the tubes array.
- The INIT_ENTRY subcommand causes the INIT_TUBES subroutine to be run at registration time. This initializes the tubes array once for all processes.

Maps and Addresses

8

PRIMOS provides a variety of information that you can use to locate and examine EPFs currently mapped to memory. This information can be helpful when you debug a dynamic EPF. The basic procedure is as follows:

- Use the LIST_EPF command to find the actual addresses of the procedure, linkage/data, and other areas of the EPF.
- Use the map generated with the MAP subcommand of BIND to find the imaginary addresses of specific items in the EPF, such as the Entry Control Block (ECB), program base, and linkage base of each procedure.
- Correlate these imaginary addresses with actual addresses.
- Examine the EPF in memory using one of the Prime debuggers, VSPD and ISPD.
- Use the DUMP_STACK command to find the locations of your EPF's stack frames.

Imaginary and Actual Addresses

Before you begin, you need to understand how PRIMOS correlates imaginary and actual addresses. Recall that EPFs are built with imaginary addresses so that PRIMOS can locate them in any available memory space. Both imaginary and actual addresses have the form

segment_number/offset

You can identify imaginary addresses because they have signed segment numbers. For example,

```
–0002/10472
+0000/77160
–4/1672
+6/1000
```

Actual addresses have unsigned segment numbers.

```
4376/15433
4377(0)/100123
```

Note Both imaginary and actual address segment numbers may be displayed with or without leading zeros. Actual addresses may also be displayed with ring numbers in parentheses following the segment number. In the last example, the ring number 0 is shown. The ring numbers are not relevant to the current discussion.

Signed Segment Numbers

BIND chooses the signs of imaginary segment numbers as follows:

- BIND assigns positive segment numbers to pure procedure segments in dynamic EPFs.
- BIND assigns negative segment numbers to linkage, data, and impure procedure areas in both registered EPFs and dynamic EPFs.
- BIND assigns negative segment numbers to shared procedure areas in registered EPFs.

LIST_EPF Command

The LIST_EPF command shows the names of all EPFs currently mapped to your address space. Use the –SEGMENTS option to get a listing that shows the actual address of each imaginary segment.

```
OK,  LIST_EPF -SEGMENTS

4 Process-Class Library EPFs.

(not active)
<TPLAB>LIBRARIES*>PL1_SYSTEM_LIBRARY.RUN
  1 procedure segment:    +0:4233
  2 linkage areas:          (not allocated)
  (active)
<TPLAB>LIBRARIES*>SYSTEM_LIB$PRC.RUN
  1 procedure segment:    +0:4301
  1 linkage area:         -2:4307(3)/111702
(not active)
<TPLAB>LIBRARIES*>TRANS_LIB$PRC.RUN
  1 procedure segment:    +0:4304
  1 linkage area:          (not allocated)
(active)                 TUBES_LIB.RUN
  1 procedure segment:    +0:7777
  3 linkage areas:        -2:3777(0)/220
-4:3776(0)/0
                          -6:6023(0)/0

6 Program-Class Library EPFs.

(not active)             <TPLAB>LIBRARIES*>CC_LIBRARY.RUN
  4 procedure segments:   +0:4174              +2:4173
                          +4:4172              +6:4171
  2 linkage areas:        -2:4307(3)/13702
-4:4170(0)/0
(not active)             <TPLAB>LIBRARIES*>CC_LM.RUN
  1 procedure segment:    +0:4226
  1 linkage area:          (not allocated)
(not active)
<TPLAB>LIBRARIES*>FTN_LIBRARY.RUN
  1 procedure segment:    +0:4251
  1 linkage area:          (not allocated)
(not active)
<TPLAB>LIBRARIES*>PL1_LIBRARY.RUN
  2 procedure segments:   +0:4235              +2:4234
  2 linkage areas:         (not allocated)
(not active)
<TPLAB>LIBRARIES*>PRIMOS_LIBRARY.RUN
  1 procedure segment:    +0:4275
  1 linkage area:          (not allocated)
(not active)
<TPLAB>LIBRARIES*>SYSTEM_LIB$PRG.RUN
  1 procedure segment:    +0:4302
  1 linkage area:          (not allocated)

2 Program EPFs.

(not active)             <TPLAB>CMDNC0>TERM.RUN
  1 procedure segment:    +0:4306
  1 linkage area:         -2:4307(3)/3620
(not active)             <TPUSR3>WRITER>MY_PROG.RUN
  1 procedure segment:    +0:4305
  1 linkage area:         -2:4307(3)/4014
```

The listing shows the actual locations of imaginary segments in two formats:

- Pure procedure segments are displayed as

  +imaginary_seg_num:actual_seg_num

- Linkage/data, shared procedure, and impure procedure segments are displayed as

  –imaginary_seg_num:actual_seg_num/offset

Pure procedure segment addresses are shown without offsets because they are mapped so that the beginning of an imaginary segment corresponds exactly to the beginning of an actual segment. PRIMOS maps only one of these segments to each actual segment. For example, in the above listing, imaginary segment 0 of the MY_PROG.RUN is mapped to actual segment 4305.

Linkage/data segments are shown with offsets because PRIMOS can map them anywhere within an actual segment. PRIMOS can map more than one of these imaginary segments to a single actual segment. For example, imaginary segment –2 of MY_PROG is mapped so that it begins at offset 4014 of actual segment 4307.

Impure and shared procedure segments are also shown with offsets, but the offsets are 0 since PRIMOS always maps them to begin at the beginning of an actual segment. For example, imaginary segment –4 (the shared procedure segment) of the registered library EPF named TUBES_LIB begins at offset 0 of segment 3776.

Note TUBES_LIB is shown in the display without a pathname. Dynamic EPFs are displayed by LIST_EPF with full pathnames. Registered EPFs are displayed by LIST_EPF without pathnames.

The BIND Map

The MAP subcommand of BIND generates an imaginary memory map of an EPF showing the imaginary addresses of several structures for each procedure.

The memory maps for dynamic EPFs and registrable EPFs differ in their Segment fields. The segment types for a dynamic EPF appear as follows:

```
Segment Type             Low    High    Top
 -0002  DATA            000000 000117 000332
 +0000  PROC            001000 001146 001150
```

The Segment fields for the same program as a registrable EPF appear as follows:

```
Segment Type                Low    High    Top
 -0004  DATA              000000 000051 000264
 -0002  SHARED PROC       001000 001172 001174
 +0000  PROC              177777 000000 001000 EMPTY
```

In a registrable EPF, pure procedure code and shareable linkage are placed in a separate shared data segment (SHARED PROC). The DATA segment contains static data and per-user linkage. In the above example, all procedure code and most linkage are shared, so the DATA segment is smaller than the corresponding one in the dynamic EPF, and the unshared PROC segment (for impure procedure code) is unused.

The following example shows the map of the dynamic EPF named MY_PROG.RUN:

```
Map of MY_PROG                         (Map Version 1)

  MAIN PROCEDURE: GG$MAIN
  START ECB: -0002/000017

  Segment Type              Low    High    Top
   -0002  DATA            000000 000126 000130
   +0000  PROC            001000 001207 001210

    Base Area: +0000 000100 000100 000777 000777

PROCEDURES:
  Name       ECB address  Initial PB%  Stack size   Link  size  Initial LB%
  GG$MAIN    -0002/000017 +0000/001000   000046       000130   -0002/177400
  SUBR1      -0002/000047 +0000/001013   000046       000130   -0002/177400
  SUBR2      -0002/000077 +0000/001027   000046       000130   -0002/177400

DYNAMIC LINKS:
  CC$PRINTF
             +0000/001202

COMMON AREAS:

OTHER SYMBOLS:
  CC$COP     +0000/001042

UNDEFINED SYMBOLS:
```

For the procedure SUBR1, for example, the ECB is located at offset 000047 of imaginary segment –0002; the procedure base is located at offset 001013 of imaginary segment +0000; and the link base is located at offset 177400 of imaginary segment –0002. All addresses are in octal numbers.

From Imaginary to Actual Addresses

After MY_PROG is mapped to your address space, you can combine information given in the BIND map with the information displayed by the LIST_EPF command to find the actual location of each structure. For example, suppose that LIST_EPF –SEGMENTS displays the following information about MY_PROG:

```
(active) <USRDSK>USER_PROGS>MY_PROG.RUN
  1 procedure segment:    +0:4562
  1 linkage area:         -2:4625(3)/2734
```

You can find the locations of various structures for SUBR1 as described in the following paragraph.

Determining the Procedure Base Address

The LIST_EPF display shows that segment +0 is mapped to actual segment 4562. No offset is shown because imaginary segment +0 is a procedure segment that is mapped to begin at the beginning of an actual segment. Therefore, the offset of SUBR1's procedure base in the actual segment is the same as the offset in the imaginary segment (001013). The actual address of SUBR1's procedure base is therefore 4562/1013.

Determining ECB Addresses

The BIND map shows the imaginary address of SUBR1's ECB as offset 000047 of imaginary segment –0002. The LIST_EPF display shows that imaginary segment –2 is mapped to offset 2734 of actual segment 4625. Since segment –2 is a linkage/data segment, PRIMOS need not map it to begin at the beginning of an actual segment.

To calculate the offset portion of the actual address, you need to add the ECB's offset within the imaginary segment (47) to the offset of the imaginary segment within the actual segment (2734). This gives an actual offset for the ECB address of 3003, so the actual address is 4377/3003.

Note When doing arithmetic with offsets, remember that they are octal numbers. Note that the highest offset number in a segment is 177777 and that there is no carry into the segment number. For example, 177777 + 1 = 000000. The next section shows how you can have one of the Prime symbolic debuggers do the arithmetic for you when you are examining an EPF in memory.

Determining the Link Frame Address

To calculate the address of the link frame, you need to remember that the address placed in the link base register is always 400 less that the actual address of the link frame. The address shown in the column headed `Initial LB%` in the BIND map is the imaginary equivalent of the value placed in the link base register.

You can calculate the actual address placed in the link base register just as you calculate the address of an ECB. To calculate the offset portion, you add the offset within the imaginary segment to the offset of the imaginary segment within the actual segment. In this case, you add 177400 and 2734 to get an actual offset of 2334 (following the offset arithmetic rules given above). Therefore, the actual address placed in the link base register is 4625/2334.

To get the actual location of the link frame in memory, add 400 to the offset calculated above. This gives an actual address for the link frame of 4625/2734. Note that in this case the calculation is trivial since the link base for the procedure is at the beginning of the linkage segment (177400 + 400 = 0). This is frequently the case, so you can often avoid doing any calculations and simply read the location from the LIST_EPF display.

Examining EPFs in Memory

You can examine a dynamic EPF in memory using one of the Prime interactive debugging utilities, VPSD and IPSD. VPSD handles V-mode code and IPSD handles I-mode code. This section briefly shows you how to use VPSD. The IPSD subcommands are nearly identical. The *Assembly Language Programmer's Guide* gives complete information about both debuggers.

Examining Mapped EPFs

To examine a dynamic EPF that is suspended or that has run to completion but remains mapped, invoke the debugger with the VPSD command.

When you use VPSD to examine a program already in memory, you can use the information provided by the LIST_EPF command and the BIND map to locate EPF structures in memory. To examine an object at a given imaginary location, you need three pieces of information from the imaginary and actual addresses of the object:

- The actual segment number
- The offset of the imaginary segment within the actual segment (relocation offset)
- The imaginary offset

You can have VPSD display the location using the following subcommand sequence:

SN *actual_segment_number*
RE *relocation_offset*
A > *imaginary_offset*

For example, the BIND map shows that the ECB of SUBR_1 begins at imaginary address –0002/000047. The LIST_EPF display shows that imaginary segment –2 is mapped to actual address 4625/2734, so you can access the first location of the ECB as follows:

```
OK, VPSD

[VPSD Rev.T1.1-21.0 Copyright(c)Prime Computer,Inc. 1988]

$:O        tells VPSD to display locations in octal format

$SN 4625                      set the actual segment number

$RE 2734                          set the relocation offset

$A >47                              access imaginary offset
4377/>47 4562                  VPSD shows the actual address
                                    followed by its contents
                       (>47 means the relocation offset + 47)
```

For imaginary addresses with positive segment numbers, the relocation offset is 0, so you can eliminate the RE subcommand. For example, the first executable instruction of SUBR1 is at imaginary address +0000/001013 and segment +0 is mapped to actual segment 4562. You can then examine the code at this address as follows:

```
$:S                      tells VPSD to display locations as
                         symbolic instructions
$SN 4562

$A 1013
4562/ 1013 EAL% LB%+ 467       You press Return several
```

```
4562/ 1015 STL% SB%+ 42        times to display a number
4562/ 1017 PCL% LB%+ 411,*     of locations
4562/ 1021 AP   SB%+ 42,SL
```

You can also use VPSD's virtual base registers to hold values for the link base, procedure base, stack base, and XB . You can then address locations relative to these values without doing any further arithmetic. To set the LB register, for example, use the subcommand format:

LB *actual_segment relocation_offset+imaginary_offset*

You can set the other virtual registers in a similar manner.

For example, suppose you wish to examine the Indirect Pointer addressed by the PCL instruction at offset 1017 shown above. Since the imaginary address of SUBR_1's LB is –0002/177400, the sequence is as follows:

```
$:O

$LB 4625 2734+177400               Set the virtual linkage
                                   base register

$A LB%+411                 Use the relative address shown
                           in the PCL instruction
4625/ 2745 104562
4625/ 2746 1202
```

In this case, the display shows a faulted IP (high-order bit set) that points to a dynt in the procedure segment.

Using the DUMP_STACK Command

Use the DUMP_STACK command to display the addresses of stack frames allocated for your program. The display may show many frames, so identification of the desired frame may not be 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 PL/I, F77, VRPG, or Pascal.
- By comparing the (LB=) field to the value calculated using the LIST_EPF display and the BIND map. This works for procedures written in any language.

In the first case, the name of the owning procedure appears in the DUMP_STACK display itself.

In the second case, you need to do some address calculations. First, you determine the Initial LB% for the procedure, as explained above. Then you look for the corresponding stack frame in a DUMP_STACK display. If it isn't there, the procedure you are searching for is not active, and is therefore not on the stack.

Locating the Stack Frame for a Procedure

Here is a sample display from the DUMP_STACK command. It shows a running program EPF that was interrupted by pressing 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 your 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 PL/I 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 a PMA subroutine it called is 6002/3710. You pressed CONTROL–P while this PMA subroutine was executing, signalling the QUIT$ condition.

In more complicated situations, the stack frames are often not so easy to identify, so comparison with the LB registers displayed for each frame is 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 the debugger and examine the calling instruction to determine the address of the ECB. Often, the calling instruction is a PCL instruction that makes an indirect reference through an IP in the link frame (LB-relative). If this is the case, you must also set the value of LB in the debugger 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 the debugger 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. Look 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, 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 you that the instruction that called the SUBR1 procedure is at address 41/125336.

Now 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 you that the SB for the calling procedure is 6002/3462 and the LB for it is 41/125010. Now, enter the debugger, examine the calling instruction, and track down its IP to the ECB. Once you have the actual ECB address, use LIST_SEGMENT to show which EPFs have linkage in that segment. Then use LIST_EPF -SEGMENTS to determine whether there is a match between the ECB address, the actual address of the linkage 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      [press Return to continue]
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 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 procedure invocation, you can reenter the debugger and examine the stack frame with the following commands:

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.

Note If the stack switches to another segment, the new segment number is shown in the DUMP_STACK display at that point. Watch for `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.

9 EDIT_BINARY

EDIT_BINARY is an interactive tool that allows you to create and edit binary libraries. You can use EDIT_BINARY to create a binary file containing the linkage information for a library EPF. A binary file of this type is known as a **binary library**. You can also use EDIT_BINARY to edit or combine existing binary libraries. EDIT_BINARY generates a new binary file as output. You use this binary file when building an EPF using BIND.

When creating a binary library, EDIT_BINARY reads the entrypoint table of a library EPF and creates a binary library containing a dynamic link for each entrypoint. The library EPF can be either a dynamic EPF or a registrable EPF. You can also use EDIT_BINARY to create individual dynts, load in other binary files, list the entrypoints in a binary library, include bypass information, and carry out a variety of other useful functions.

The syntax of the EDIT_BINARY command is as follows:

EDIT_BINARY *output_pathname* **-DAM -SKIP -NO_QUERY -HELP**

The command options are described later in this chapter.

The EDIT_BINARY command returns a colon (:) as a prompt character. You respond to this prompt by issuing EDIT_BINARY subcommands, ending the session with either FILE or QUIT. The command interface is very similar to the BIND command interface, and EDIT_BINARY and BIND subcommands with similar functions have the same names. If you are familiar with BIND, you should have no difficulty using EDIT_BINARY.

Note If you have created your own binary libraries in the past using EDB, you should find that EDIT_BINARY is a much more efficient tool. For information on Prime's other binary editing tools, see Appendix B.

This chapter shows you how to create a simple binary library with dynamic links corresponding to the entrypoints in a library EPF. The chapter also includes a complete reference to EDIT_BINARY subcommands and options that you can use for creating more complex binary libraries.

Creating a Binary Library From a Library EPF

You need only a few EDIT_BINARY subcommands to create a binary library with dynamic links corresponding to the entrypoints in a library EPF. The following sample session shows you how.

Suppose that you have created a library EPF called MY_LIBRARY.RUN that includes 10 entrypoints. To create a binary library called MY_BINARY.BIN with dynamic links to each of these subroutines, use the following sequence of commands:

```
OK, EDIT_BINARY MY_BINARY.BIN
[EDIT_BINARY Rev.T3.0-23.0 (c)1990, Prime Computer, Inc.]
: RFL
: READ MY_LIBRARY.RUN
  Library creation date:        90-10-16.10:01:16.Tue
  Library type:                 program
  Number of entries:            10
End processing MY_LIBRARY.RUN.
: SFL
: FILE
OK,
```

The following comments explain the sample session:

- The EDIT_BINARY command line specifies the output filename, MY_BINARY. If you don't specify the filename here, you must give it at the end of the session with the FILE subcommand.
- The RFL subcommand adds code to the binary library to tell BIND to load only those modules that it needs to resolve references. This reduces the size of the EPF runfile by preventing the inclusion of unneeded modules. Most binary libraries should begin with an RFL and end with an SFL.
- The READ subcommand reads the entrypoint table of the specified library EPF, creating a dynt for each entry and reporting on the number of EPF library entries found.
- The SFL subcommand complements the RFL subcommand by telling BIND to load all subsequent modules whether they are needed or not. If your binary library build sequence includes an RFL, you should follow it with an SFL. This assures that any binary files you load after your library during a BIND session are loaded correctly.

You can use EDIT_BINARY to create more complex libraries in order to simplify your linking tasks. For example, you can have EDIT_BINARY read more than one library EPF in order to create a binary library with dynamic links to several runtime libraries. You can also include references to Prime-supplied binary libraries in your binary libraries.

For example, suppose you have created a library EPF of language-specific routines called MY_CC_LIBRARY.RUN which you call from C language programs. By following the procedure given in the previous example, you could create a binary library, called MY_C_LIB.BIN, containing dynamic links to the entrypoints in MY_CC_LIBRARY.RUN. A typical BIND session using this binary library might look like this:

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LO MY_C_PROGRAM.BIN
: LO MY_C_LIB.BIN
: LI C_LIB
BIND COMPLETE
: FILE
OK,
```

You can simplify this build sequence by including a reference to the Prime-supplied binary library C_LIB in your custom binary library. The following example shows you how.

```
OK, EDIT_BINARY
[EDIT_BINARY Rev.T3.0-23.0 (c)1990, Prime Computer, Inc.]
: RFL
: READ MY_CC_LIBRARY.RUN
  Library creation date:        90-10-18.11:03:28.Thu
  Library type:                 process
  Number of entries:            34
End processing CC_LIBRARY.RUN.
: LIBRARY LIB>C_LIB
: SFL
: FILE MY_C_LIB.BIN
OK,
```

The LIBRARY command adds code to your binary library that tells BIND to load another binary library. In this case, your binary library MY_C_LIB would reference the LIB > C_LIB binary library. This is equivalent to typing LIBRARY C_LIB during the BIND session. You can now build your C program with BIND by loading only MY_C_LIB.

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LO MY_C_PROGRAM.BIN
: LO MY_C_LIB.BIN
BIND COMPLETE
: FILE
OK,
```

In effect, you have made all the dynamic links in the Prime-supplied binary library C_LIB available from your binary library. If you frequently need to load a Prime-supplied library along with your own binary library, you can simplify your build sequences by associating the binary libraries in this way.

Note You can also use the LOAD subcommand of EDIT_BINARY to load the actual text of a Prime-supplied library EPF into your binary library. However, this creates a much larger binary file. It also complicates library maintenance, because you would need to rebuild your binary library whenever you wished to incorporate a new version of the Prime-supplied library EPF.

You may wish to set your BINARY$ search rules to enable EDIT_BINARY and BIND to locate binary files. For further details on search rules refer to the *Advanced Programmer's Guide II: File System.*

EDIT_BINARY Reference

This section explains all EDIT_BINARY command line options and subcommands.

Command Line Options

Option	*Description*
output_pathname	Used optionally to specify the pathname of the output binary library file to be created. The file is automatically assigned the .BIN suffix; specifying the suffix on the command line is optional. You can specify a complete pathname or a filename; if you specify a filename, EDIT_BINARY creates the file in your currently attached directory.

	You can specify the output pathname here, or within EDIT_BINARY when you issue a FILE subcommand. EDIT_BINARY prompts you to resolve discrepancies if you specify different output pathnames in these two places or specified no filename at all.
-DAM	Causes EDIT_BINARY to create a DAM (Direct Access Method) output file. The default is SAM.
-HELP	Provides on-line help. This option displays the correct command line syntax, then returns to PRIMOS command level without executing the EDIT_BINARY command.
-NO_QUERY **-NQ**	Suppresses queries that occur • When you give the QUIT subcommand without saving the file created • When you give a FILE subcommand that would overwrite a file with the same name -NO_QUERY does not suppress queries to resolve descrepancies in filenames supplied during the EDIT_BINARY session.
-SKIP	Causes EDIT_BINARY to include bypass information with each module loaded. This speeds up linking by allowing BIND (or SEG) to skip unneeded routines efficiently instead of reading and discarding all unwanted object text.

Subcommands

CHANGE_SYMBOL_NAME *old_entryname new_entryname*
CSN

Places code in your file that causes BIND (and SEG) to change all previous occurrences of *old_entryname* to *new_entryname.*

COMMENT
CMT

Defines the beginning of a non-executing comment in the output binary file. The comment continues until you insert a line of text that contains only a dollar sign ($) character. A comment may be up to 508 characters in length. EDIT_BINARY truncates comments that exceed this maximum length.

```
COPY  [ -UNTIL entryname     ]
C     [ -INCLUDING entryname ]
      [ -END                 ]
```

Copies information from the current file position of the input binary file to the output binary file. The subcommand options define where copying ends. The -UNTIL option stops the copy operation when the module containing the named entry is encountered.

The -INCLUDING option stops copying after copying the module containing the named entry. The -END option copies until it encounters the end of the input file. If you specify no option, COPY copies until it encounters the end of the file. You must use the OPEN subcommand to open the input file before issuing the COPY subcommand. You can use the LOCATE subcommand to change the current file position within the input file.

```
DEFAULT_DYNT_TYPE  { -PER_USER }
DDT                { -SHARED   }
                   { -DEFAULT  }
```

Sets the default dynt type used by subsequent DYNT subcommands. This default applies if you issues a DYNT subcommand with either no option or the -DEFAULT option. Initially, EDIT_BINARY has a default dynt type of per-user. You can issue the DEFAULT_DYNT_TYPE command at any time during an EDIT_BINARY session to redefine the default dynt type. Dynt types are discussed in greater detail in Chapter 6.

```
     ┌ -PER_USER ┐
DYNT │ -SHARED   │ entryname [ entryname...]
     └ -DEFAULT  ┘
```

Defines *entryname* as a dynamic link (dynt). You can specify more than one *entryname* to define multiple dynamic links. Use the -PER_USER, -SHARED, and -DEFAULT options to set the dynt type. If you don't specify an option, the dynt type is set to the option specified by the most recent DEFAULT_DYNT_TYPE subcommand. If no DEFAULT_DYNT_TYPE subcommand has been issued during this EDIT_BINARY session, the dynt type is set to per-user. Dynt types are discussed in greater detail in Chapter 6.

EXTRACT *entryname*
E

Extracts the module containing the specified entryname from the input library EPF and copies it to the output binary library. You must first open the input file by issuing the OPEN subcommand. The EXTRACT subcommand permits you to extract multiple entries by listing entrynames. Entries are searched for in the order listed; each search begins at the current file position. If an entryname cannot be found, EXTRACT returns an error message and resets the current file position to the end of the input file. It does not search for subsequent listed entrynames.

FILE [*pathname*]

Saves the binary library generated during the EDIT_BINARY session. *pathname* specifies the name under which the file is to be saved. If you don't supply *pathname*, EDIT_BINARY does one of the following:

- Saves the file under the pathname you supplied when you invoked EDIT_BINARY
- Prompts you for a pathname if you did not supply one when you invoked EDIT_BINARY

If you supply different pathnames with the FILE subcommand and on the EDIT_BINARY command line, EDIT_BINARY asks you to resolve the difference.

```
HELP { -LIST         }
     { command_name  }
```

Displays information on EDIT_BINARY subcommands. Use the -LIST option to get a list of EDIT_BINARY subcommands. Give a *command_name* to get help on a specific subcommand.

LIBRARY *pathname*
LI

Includes code in your binary library that causes BIND (or SEG) to load the additional binary library specified in *pathname*.

```
              [ -COMMENTS      ]
              [ -COMMONS       ]
              [ -DYNTS         ]
              [ -ENTRIES       ]
LIST_CONTENTS [ -LIBRARIES     ]
LC            [ -ALL           ]
              [ { -END      }  ]
              [ { entryname }  ]
              [ -BY_MODULE     ]
              [ -NO_WAIT       ]
```

Displays the contents of the input binary library. You must first open the input file using the OPEN subcommand.

You can specify options to list comments (-COMMENTS), common areas (-COMMONS), dynt names (-DYNTS), entrynames (-ENTRIES), library names (-LIBRARIES), or all of these items (-ALL). Multiple options can be specified in any sequence.

LIST_CONTENTS indicates dynt types with the abbreviations PU (per-user), SH (shared), and DF (default). It also uses the abbreviations C (common area), ENT (entry), and LIB (library).

LIST_CONTENTS lists the items from the current file position to either the end of the file (specified by the -END option) or until it encounters the specified *entryname*. If no option is specified or the *entryname* is not found, LIST_CONTENTS lists to the end of the input file and resets the current file position to that point. To return to the beginning of the input file, issue the TOP subcommand.

If you specify the BY_MODULE option, LIST_CONTENTS groups entrynames together according to their compilation units.

The NO_WAIT option scrolls out the item listings without pausing for user response after each screenful of information.

LOAD *pathname* [*pathname*...]
LO

Includes the specified binary files in your output binary file. If you give a filename instead of a pathname, BINARY$ search rules are used to find the file.

LOCATE
LOC $\left\{\begin{array}{l}\textbf{SFL}\\ \textbf{RFL}\\ \textit{entryname}\end{array}\right\}$

Moves the current file position in the input binary file to the next SFL or RFL group, or to the beginning of the compilation unit that contains the specified entryname. You must first open the input file using the OPEN subcommand. If the location cannot be found, LOCATE sets the current file position to the end of the file and issues an error message.

OPEN *pathname*

Opens a binary file for use as input. You must open an input file before issuing a COPY, EXTRACT, LIST_CONTENTS, LOCATE, or TOP subcommand. EDIT_BINARY only permits one input file to be open at a time; opening an input file automatically closes any previously open input file. If you give a filename instead of a pathname, BINARY$ search rules are used to find the file.

QUIT
Q

Ends the EDIT_BINARY session.

Caution EDIT_BINARY does not save the binary file created during the EDIT_BINARY session when you issue the QUIT subcommand. To save the file, you must issue the FILE subcommand. The FILE subcommand saves the file and quits the session; the QUIT subcommand just quits the session.

```
READ epf_pathname  [ -PER_USER ]
RE                 [ -SHARED   ]
                   [ -DEFAULT  ]
```

Reads the library EPF specified by *epf_pathname* and places a dynt in your binary file for every entryname found in the library EPF entrypoint table.

The *epf_pathname* can be a dynamic EPF or a registrable EPF. You specify a registered library EPF by supplying the pathname of the registrable EPF file; you cannot read a registered library EPF directly. You can read in a registrable EPF file before or after you have actually registered the library EPF.

The binary file created by this subcommand has a dynt type assigned to all of its dynts. You can specify the dynt type using the -PER_USER, -SHARED, or -DEFAULT option. If you don't specify a dynt type option in this subcommand, the dynt type created depends on the library EPF type.

- If the library is a dynamic EPF, per-user dynts are generated in the binary library.
- If the library is a registrable EPF, default dynts are generated in the binary library.

Use the -PER_USER, -SHARED or -DEFAULT option to override these dynt type defaults. Dynt types are discussed in greater detail in Chapter 6.

RFL

Writes code for a reset-force-load (RFL) to your binary file. BIND only loads binary modules that are required to resolve outstanding references while RFL is in effect. The RFL remains in effect until BIND encounters set-force-load (SFL) code. Most binary libraries begin with an RFL. If you begin your library with an RFL, be sure to end it with an SFL.

SFL

Writes code for a set-force-load (SFL) to your binary file. BIND loads all binary modules, whether they are required to resolve a reference or not, while SFL is in effect. SFL is in effect by default until BIND encounters an RFL.

TOP

Positions the current file pointer to the top of the input binary file. You must first open the input file using the OPEN subcommand before issuing this subcommand.

Other binary editing tools are described in Appendix B.

Appendices

Coding EPFs in PMA

A

This appendix 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 consists of one or more **modules**. A module may contain one or more **subroutines**. When a module is assembled (using PMA), an object file is generated, usually with the .BIN filename suffix. This object file 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 a module you are building by including special **pseudo-operations** in the PMA source code. Pseudo-operations are directives to the assembler; usually, they change the way in which subsequent lines in the source file are interpreted. Pseudo-operations themselves may or may not cause specific data (such as instructions or storage allocation information) to be generated in the object text.

The last line of all PMA modules must be an END pseudo-operation. Usually there is only one module in a source file, but it is possible to create a source file containing multiple modules, each module terminated with an END pseudo-operation. 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-operation. No comment lines or blank lines may follow the END pseudo-operation. Other important pseudo-operations 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-operation to specify a V-mode or I-mode module
SYML	Optional pseudo-operation to turn on long (as many as 32 characters) symbol names
RLIT	Optional pseudo-operation to cause placement of literals in procedure text
ENT	Optional pseudo-operations to export names for reference by external modules
LINK	Pseudo-operation to switch to linkage text generation for placement of the ECB
ECB	Pseudo-operation to generate the ECB itself, and, optionally, additional ECBs for alternate entrypoints or internal subroutines
DYNM	Pseudo-operation to specify stack frame allocation
EXT	Optional pseudo-operations to specify external symbols
PROC	Pseudo-operation to switch back to generating procedure text
instructions	The procedure code of the module
LINK	Optional pseudo-operation for switching to linkage text generation
data	Various address definition, data definition, and storage allocation pseudo-operations (optional), to describe the format and data for the link frame
END	Pseudo-operation 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 50 Series machines, see the *System Architecture Reference Guide.*

Use of SEG or SEGR

The first non-comment line of a PMA subroutine must be either the SEG pseudo-operation (for a V-mode subroutine) or the SEGR pseudo-operation (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.

If you are creating a registered EPF, you should also specify the –SPLIT_LINKAGE option with the SEG or SEGR pseudo-operations. This option causes the assembler to separate the linkage from the data and store them in separate frames. It is advisable to reassemble existing programs with the –SPLIT_LINKAGE option before using them to build registered EPFs. If you specify –SPLIT_LINKAGE, BIND can create a registered EPF that shares linkage. If you do not specify –SPLIT_LINKAGE, BIND can create a registered EPF, but one that cannot share linkage.

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-operation

```
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-operation

```
LINK
```

Stack and Parameter Allocation Information

The DYNM pseudo-operation 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-operation 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 of the argument list template in this case is *argument–1*, and the number of arguments is 4.

The DYNM pseudo-operation provides the only way of allocating stack frame storage in PMA. Using an EQU pseudo-operation 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 a dynamic 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, all ECBs in dynamic EPFs should be placed in the linkage text. When you are creating a registrable EPF with SPLIT_LINKAGE, BIND places ECBs from the linkage text in shared linkage. With registered EPFs you can also place ECBs in the procedure text.

ECBs are created with the ECB pseudo-operation

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

Note that the ECB is placed in the linkage text for the module. The ECB is labeled with *ecb_label*, which identifies the actual target of procedure call (PCL) instructions to the entrypoint. The ENT pseudo-operation 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-operation). The number of arguments is specified as *n_args*.

The ECB pseudo-operation 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-operation. Another optional field, the stack size, defaults to the amount of stack space explicitly reserved via the DYNM pseudo-operation. 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-operation

```
CALL subroutine
```

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

1. Identifies *subroutine* as an external reference, as if the pseudo-operation EXT *subroutine* had been issued
2. 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 pseudo-operations sequence had been present:

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

3. 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-operation 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-operation 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 XAC pseudo-operation can be used to link to a procedure segment or an impure segment label. However, you must supply the segment number independently.

The COMM pseudo-operation 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 referenced location in a common area. This method produces more efficient code in terms of execution time, at the expense of the size of linkage text (because more than one IP may be generated for each common area). By using this method, PMA avoids use of the XB or X register — registers that 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 (or 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-operation 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 by using the following command sequence:

```
OK, BIND
[BIND Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
: LOAD failing-program.RUN
: MAIN main-entrypoint-name
BIND COMPLETE
: FILE working-program.RUN
OK,
```

Restrictions on Writing PMA Modules for EPF Execution

When writing a module in PMA for execution within an EPF, several restrictions must be observed. This section discusses these restrictions.

- 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.

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-operation 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.

The following coding practices result in impure PMA modules:

- Self-modifying code
- An ECB in the procedure text
- An IP in the procedure text
- A JST instruction to an internal subroutine
- An RLIT or FIN pseudo-operation, when storing into a literal

These coding practices are described in the following paragraphs.

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, if you place an ECB in the procedure text of a dynamic EPF, the procedure text is no longer pure. The actual placement of the linkage text is determined when the program is run, not when it is linked by BIND. 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.

When building a dynamic EPF, BIND issues a warning message if it finds an ECB in procedure text, unless you have specified the IMPURE keyword with the SEG or SEGR pseudo-operations. If the resulting EPF is executed it may produce an access violation error when the offending module is invoked. This access violation is invoked because the imaginary address has not been translated into an actual address.

Similarly, placing IPs (Indirect Pointers) in the procedure text of a dynamic EPF normally results in an impure module. However, there is a way to create pure procedure text containing IPs. When you use BIND to build the dynamic EPF, use the SYMBOL subcommand to explicitly locate the external symbols referenced by the IPs.

A JST (Jump and STore) instruction that references an internal subroutine also produces impure code. This is 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-operations 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-operation 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, then the SEG or SEGR pseudo-operation at the top of the module should read SEG PURE or SEGR PURE. The SEG and SEGR pseudo-operations default to PURE if you do not specify a keyword. However, explicitly specifying the keyword PURE can be a convenient signal to other programmers that you have checked the module for purity. If this convention is used, then any PMA module without a PURE or IMPURE keyword following the SEG or SEGR pseudo-operation should be checked for purity before being linked into an EPF.

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
```

Storing Into IPs or ECBs

If your program declares Indirect Pointers (IPs) or Entry Control Blocks (ECBs), you must be careful about modifying them during execution. For example, consider the following subroutine in a dynamic EPF:

```
          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
*
          PRTN
*
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. There are two exceptions to this rule. THE_IP would be reinitialized if the program is removed from memory. THE_IP would also be reinitialized if the K$INIT_ALL key is supplied to EPF$INIT by a user program. The EPF$INIT subroutine is further described in the *Advanced Programmer's Guide III: Command Environment.*

For example, if you call this subroutine from a program that simply calls TESTSUBR once and then exits, THE_IP would not be reinitialized after each call. Invoking the program repeatedly would not produce identical results, 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, removes the EPF from memory. This forces the complete reinitialization of the EPF at the next RESUME command, and thus restores 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 information (using the BSS pseudo-operation) 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 information 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 information 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,
```

When you build a registered EPF in which you have specified the SPLIT_LINKAGE keyword, BIND places all IPs created with the IP pseudo-operation in shared linkage. Such IPs can never be modified under any circumstances. If you wish to create a modifiable IP, use the MIP pseudo-operation. IPs created with MIP are placed in per-user linkage, so they may be modified.

Similarly, you cannot assume adjacency of IPs in shared linkage. BIND may in fact move the IPs. For example, a PMA program may set up several IPs contiguous in the data frame to be used as a jump table to a set of routines or common areas. One would set up an index into this table of IPs to find the desired object. This type of table must remain in the per-user linkage to guarantee adjacency of its IPs. Use the MIP pseudo-operation to create these IPs.

Obsolete Binary Editors

B

This appendix describes the Binary Editor (EDB) and the Library Editor (LIBEDB). Both of these editors have been replaced by the more powerful EDIT_BINARY binary editor. EDIT_BINARY is described in Chapter 9. EDB and LIBEDB are still supplied with PRIMOS, but their continued use is discouraged.

EDB is used to create and modify binary libraries. LIBEDB is used once a binary library is created to decrease linking or loading time. Both of these programs operate on object code files 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.

LIBEDB

The LIBEDB program is used for editing bypass information into library files. The BIND 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, BIND 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 LIB directory. You should use LIBEDB on a library file after the library's 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. A sample LIBEDB session is shown at the end of this appendix. LIBEDB asks for an input and output filename and for a file type. In theory, a library containing 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

```
    ⎧input_file⎫
EDB ⎨- ASR     ⎬ [output_file]
    ⎩- 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 filename 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 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 entrypoint 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.

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.)

```
        ┌ name    ┐
COPY    │ ALL     │
C       │ < 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.

```
        ┌ name    ┐
FIND    │ ALL     │
F       │ < 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 entrynames 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
Q

Closes all files and exits to PRIMOS.

REPLAC *name pathname*
R

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
T

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 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]
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]
O

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. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
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. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
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 entrypoint ENT3G with the module in NFILE3.BIN containing entrypoint ENT3F and ENT3G. The output file is LIBNEW.BIN.

```
OK, EDB NFILE3.BIN
[EDB Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
ENTER, FIND ALL
ENT3F                             ENT3G

.BOTTOM.
ENTER, QUIT
OK, EDB LIBEXP.BIN LIBNEW.BIN
[EDB Rev. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
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. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
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. T3.0-23.0 (c) 1990, Prime Computer, Inc.]
SOURCE FILE, DESTINATION FILE, PARAMETER
WHERE: PARAMETER = 0 - DESTINATION FILE SAM
       PARAMETER = 2000 - DESTINATION FILE DAM
$ LIBEXP.BIN,FAST_LIBEXP.BIN,0
OK,
```

C EPFs and Static-mode Applications

If you are still maintaining static-mode applications on your system, this appendix contains useful information about two issues:

- Maintaining static-mode applications in a dynamic environment
- Converting static-mode applications to EPFs

In general, Prime recommends that you convert static-mode applications to EPFs in order to take full advantage of the flexible environment and ease of maintenance you get with EPFs. With the introduction of registered EPFs, all the performance and functionality of shared static-mode programs is now available with EPFs.

Static-mode Applications in a Dynamic Environment

With the introduction of EPFs at Rev 19.4, PRIMOS was enhanced to provide a dynamic command environment that takes full advantages of the capabilities of EPFs. Although Rev. 19.4 and later versions of PRIMOS are highly backward compatible, you need to be aware of two issues if you run static applications under these later revisions:

- Restrictions on static-mode invocations
- Restrictions on static-mode applications that share IPs

Restrictions on Static-mode Invocations

The dynamic command environment stores information about the state of multiple program invocations on the command stack. Since EPFs are placed in memory dynamically, newer invocations do not overwrite older invocations. As a result, PRIMOS can use the state information saved on the command stack to reinvoke suspended EPFs.

Suspended static-mode programs are also maintained on the command stack. However, PRIMOS does not dynamically allocate memory for static-mode programs, so newer invocations may overwrite old ones. Therefore, your ability to restart suspended static-mode applications is more restricted than with EPFs.

You encounter this restriction when you suspend a static-mode application and then invoke another static-mode application. If you later try to reinvoke the first invocation, PRIMOS displays the error message

```
Attempt to proceed to overwritten program image.
```

Once a static-mode program has been overwritten in this way, it cannot be recovered.

This is generally a problem for naive users of static-mode programs since they can invoke and suspend static-mode programs and EPFs in exactly the same way. Such users may not be aware that some of the programs they are invoking are static-mode programs and may be surprised to find that they cannot always be started. As a programmer you should be aware of this restriction if you maintain static-mode programs on a system. You should consider converting your static-mode applications to EPFs in order to take full advantage of the dynamic command environment.

Static-mode Programs That Share Linkage

Static-mode programs that share linkage may not run correctly under Rev. 19.4 or later PRIMOS.

System Administrators often place widely used static-mode programs in shared segments (using the SHARE command). Normally, such shared programs place pure code in the shared segments and impure linkage and data in per-user static segments. The linkage and data are not shared because each user needs a private copy that can be changed by the user's invocation of the program without affecting other users' invocations. Such programs should run without problems on Rev. 19.4 or later systems.

Some static-mode programs also share part of their linkage. Such programs place faulted IPs in shared rather than per-user segments. IPs can be shared in this way as long as they reference routines with fixed locations that are the same for all users, such as PRIMOS entrypoints and subroutines in shared static-mode libraries. Since they reference routines that have the same addresses for all users, these IPs don't need to be snapped separately for each user. They can be placed in shared segments and snapped just once for all users.

The shared dynts in such programs are usually snapped in one of two ways:

- At startup time, after the application is shared, a special program is run that snaps the dynts.

- When the application is shared, the shared linkage segments are not protected against modification (the segment protection value is set at 700_8). This allows the first invocation of the program to snap the dynts.

Sharing linkage in this way reduces the working set of an application and may increase execution speed by avoiding dynt snapping. However static-mode programs with such shared linkage may not run correctly on Rev. 19.4 or later systems. The routines referenced by shared IPs in such programs may now be in dynamic library EPFs rather than in static-mode libraries. This can cause such programs to fail when they attempt to reference these routines.

Dynamic library EPFs do not have a fixed memory location that is the same for all users. The dynt snapping mechanism maps dynamic library EPFs into available segments in each user's private address space. These locations may be different for each user. Therefore, an IP that correctly points to a library routine in one user's address space may point to the wrong location for another user. Such an IP cannot be shared among all users. When users attempt to run a static shared programs that contain shared IPs to routines in dynamic library EPFs, they are likely to encounter illegal segment number, access violation, or pointer fault errors.

You can resolve this problem in two ways:

- Convert the static-mode program to a registered EPF.
- Modify the static-mode program so that it no longer shares faulted IPs.

Converting to a Registered EPF: If you feel that the advantages of shared linkage are beneficial to your program, you should consider converting it to a registered EPF. This is the recommended procedure. Registered EPFs give you all the performance benefits of shared static-mode applications, including shared linkage, in a fully dynamic environment. As an EPF your program is also easier to maintain and install than a shared static-mode program which may require a complex build sequence every time it is modified. The section, Converting From Static-mode Programs to EPFs, gives you general conversion guidelines. Chapter 6, Registered EPFs, shows you how to build a registered EPF with BIND.

Modifying an Application Not to Share Faulted IPs: If you want to maintain your application as a shared static-mode program and have it run correctly under Rev. 19.4 or later PRIMOS, you must rebuild it so that it does not share 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 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 the actual code of all dynamically-linked subroutines used by your application. That is, load all subroutines used by your application that reside in dynamic library EPFs directly into your application. Load the actual code of these subroutines, not dynamic links to the subroutines.

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, performance should not be reduced. 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).

Converting From Static-mode Programs to EPFs

Converting a static-mode program to an EPF is usually a straightforward process. For programs written in high-level languages, if the program was compiled in either V-mode or I-mode, you can usually convert it to an EPF simply be rebuilding with BIND. If the program is in R-mode, you must first compile it in V-mode or I-mode. Depending on the compiler, this may require source code changes. PMA programs may also require source code changes. Appendix A tells you how to write EPFs in PMA.

EPFs That Call Static-mode Libraries: If you write an EPF that calls subroutines in static-mode libraries, it cannot be suspended and reinvoked as freely as an EPF that does not call static-mode libraries. The restriction is much like the restriction on restarting static-mode programs. It occurs when you invoke an EPF that calls a static-mode library, suspend the program, and then invoke another program that calls the same library. If you attempt to restart the first program, PRIMOS displays the error message

```
Attempt to proceed to overwritten program image.
```

This restriction occurs because PRIMOS initializes the linkage area of a static-mode library once per program invocation. Since static-mode libraries are always loaded into the same area of memory, only one copy of a static-mode library can be maintained in a user's address space. This means that later invocations reinitialize the same linkage/data area that was used by earlier invocations. This corrupts the library linkage/data area of the earlier invocation. PRIMOS detects this condition and prevents the earlier invocation from being restarted.

This is mainly a problem for naive users who are not aware that they have invoked a static-mode library. As a programmer, you should try to avoid calling static-mode libraries from EPFs if you want your EPFs to take full advantage of the dynamic command environment.

Rewriting Build Sequences

Once you have a program that can be built as an EPF, rebuilding it with BIND is usually a straightforward process. You simply load the program and any required libraries as described in the *Programmer's Guide to BIND and EPFs*. If you are building a registered EPF, follow the guidelines given in Chapter 6 of this book, Registered EPFs.

If you previously built the program using a SEG build sequence, convert that sequence to a BIND build sequence. A BIND build sequence is considerably simpler than the equivalent SEG build sequence. Instruction for converting an old SEG build sequence to BIND are found in the *Programmer's Guide to BIND and EPFs*.

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.

How Static-mode Programs Use Registers

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 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.)

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* value 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.FTN 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.

D A List of Registered Library EPFs

Prime supplies the following library EPFs as registered library EPFs.

```
SYSTEM_LIB$PRC.RUN
SYSTEM_LIB$PRG.RUN
TRANS_LIB$PRC.RUN
TRANS_LIB$PRG.RUN
COMMON_ENVELOPE.RUN
FTN_LIBRARY.RUN
APPLICATION_LIBRARY.RUN
CC_LIBRARY.RUN
ANSI_CC_LIBRARY.RUN
MATRIX_LIBRARY.RUN
COBOL85_LIBRARY.RUN
CBL_LIBRARY.RUN
PASCAL_LIBRARY.RUN
PL1_SYSTEM_LIBRARY.RUN
PL1_LIBRARY.RUN
PL1G_LIBRARY.RUN
VRPG_LIBRARY.RUN
```

Index

Index

Surveys

Reader Response Form
Advanced Programmer's Guide I: BIND and EPFs
DOC10055–2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate this document for overall usefulness?

☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

____

____

____

____

____

____

3. What faults or errors in this manual gave you problems?

____

____

____

____

____

____

4. How does this manual compare to equivalent manuals produced by other computer companies?

☐ *Much better*	☐ *Slightly better*	☐ *About the same*
☐ *Much worse*	☐ *Slightly worse*	☐ *Can't judge*

5. Which other companies' manuals have you read?

____

____

Name: ____

Position: ____

Company: ____

Address: ____

____

______________________________ Postal Code: ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760