

Using OpenVOS Dynamic Linking and Shared Libraries

Stratus Technologies
R648-01

Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS TECHNOLOGIES, STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Technologies assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents (a) is the property of Stratus Technologies Bermuda, Ltd. or the third party, (b) is furnished only under license, and (c) may be copied or used only as expressly permitted under the terms of the license.

Stratus documentation describes all supported features of the user interfaces and the application programming interfaces (API) developed by Stratus. Any undocumented features of these interfaces are intended solely for use by Stratus personnel and are subject to change without warning.

This document is protected by copyright. All rights are reserved. Stratus Technologies grants you limited permission to download and print a reasonable number of copies of this document (or any portions thereof), without change, for your internal use only, provided you retain all copyright notices and other restrictive legends and/or notices appearing in the copied document.

Stratus, the Stratus logo, ftServer, the ftServer logo, Continuum, StrataLINK, and StrataNET are registered trademarks of Stratus Technologies Bermuda, Ltd.

The Stratus Technologies logo, the Continuum logo, the Stratus 24 x 7 logo, ActiveService, ftScalable, and ftMessaging are trademarks of Stratus Technologies Bermuda, Ltd.

RSN is a trademark of Lucent Technologies, Inc.
All other trademarks are the property of their respective owners.

Manual Name: *Using OpenVOS Dynamic Linking and Shared Libraries*

Part Number: R648
Revision Number: 01
OpenVOS Release Number: 17.2.0
Publication Date: March 2013

Stratus Technologies, Inc.
111 Powdermill Road
Maynard, Massachusetts 01754-3409

© 2013 Stratus Technologies Bermuda, Ltd. All rights reserved.

Contents

Preface	vii
----------------	-----

1. Introduction to Dynamic Linking	1-1
Shared Libraries	1-1
Description of a Shared Library	1-1
Shared Library File Type	1-2
Considerations	1-2
Static and Dynamic Linking	1-3
Dynamic Dependencies	1-3
OpenVOS and UNIX	1-4

2. Creating Shared Libraries and Dynamically-Linked Applications	2-1
Creating Dynamically-Linked Applications with OpenVOS	
Standard C	2-1
Using the <code>cc</code> Command to Compile Applications	2-1
Using the <code>bind</code> Command to Bind Applications	2-2
The <code>-shared</code> Argument	2-2
Command-Line Options	2-2
Shared Library Names	2-3
Symbol Resolution	2-3
Using the <code>vcc</code> Command to Compile and Bind Applications	2-3
Creating Dynamically-Linked Applications with GNU C or C++	2-4
Creating Applications with Other Languages	2-5

3. Running and Debugging Dynamically-Linked Applications	3-1
Running Dynamically-Linked Applications	3-1
Understanding the Dynamic Linker	3-2
Searching For and Opening Shared Library Files	3-2
Loading Shared Libraries into Memory	3-2
Resolving Symbol References	3-4

Relocating Shared Library Addresses	3-4
Handling Initialization/Termination Code	3-4
The Application Runtime Environment	3-5
Debugging Dynamically-Linked Applications	3-6
Using <code>debug</code>	3-6
Using <code>gdb</code>	3-6
The <code>LD_DEBUG</code> Environment Variable	3-7

4. The <code>dl*</code> Functions	4-1
The <code>dlopen</code> Function and Shared Library Groups	4-1
The <code>dlclose</code> Function	4-3
The <code>dlsym</code> Function	4-3
The <code>dladdr</code> Function	4-3
The <code>dlerror</code> Function	4-3

5. Shared Library Advanced Topics	5-1
Binder Shared Library Search Rules	5-1
Explicitly Named Input File Searches	5-2
Archive File Searches	5-2
Searches Specified by the <code>-lname</code> Option	5-2
Searches Done After Bind File and Command-Line Processing	5-4
Default Shared Library Searches	5-4
Implicit Shared Library Searches	5-4
OpenVOS Object Library Searches	5-5
Dynamic Linker Runtime Search Rules	5-5
The Binder <code>-soname</code> Option and File-Level Versioning	5-5
Binding Shared Libraries	5-6
The Binder <code>--as-needed</code> Option	5-7
Symbol Visibility in Shared Libraries	5-8
Controlling Visibility from <code>gcc</code> and <code>g++</code>	5-8
Controlling Visibility in the Binder Control File	5-9
The <code>-version-script</code> Binder Option	5-9
Shared System Libraries	5-10
The <code>LD_PRELOAD</code> Environment Variable	5-11

Appendix A. Troubleshooting Dynamically-Linked Applications	A-1
Dynamic Linker Cannot Find a Shared Library	A-1
Undefined Symbol Error When Library Is Opened by <code>dlopen</code>	A-1

Index

Index-1

Figures

Figure 1-1. Shared Library Dependency Tree	1-4
Figure 3-1. Shared Library Breadth-First Load Order	3-3

Preface

The manual *Using OpenVOS Dynamic Linking and Shared Libraries* (R648) documents how to create and run shared libraries and dynamically-linked applications.

This manual is intended for programmers and application developers.

Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual. The following information has been added:

- Information about [specifying neither](#) the `-Bdynamic` nor the `-Bstatic` binder option
- Information about “[The Binder --as-needed Option](#)” on [page 5-7](#)

Related Manuals

See the following Stratus manuals for related documentation.

- *OpenVOS Commands Reference Manual* (R098)
- *OpenVOS POSIX.1 Reference Guide* (R502)
- *GNU Tools for OpenVOS: User's Guide* (R453)

Notation Conventions

This manual uses the following notation conventions.

Warnings, Cautions, and Notes

Warnings, cautions, and notes provide special information and have the following meanings:



WARNING

A warning indicates a situation where failure to take or avoid a specified action could cause bodily harm or loss of life.



CAUTION

A caution indicates a situation where failure to take or avoid a specified action could damage a hardware device, program, system, or data.

NOTE

A note provides important information about the operation of a Stratus system.

Typographical Conventions

The following typographical conventions are used in this manual:

- Italics introduces or defines new terms. For example:

The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

```
change_current_dir (master_disk)>system>doc
```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

```
list_users -module module_name
```

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

```
display_access_list system_default
```

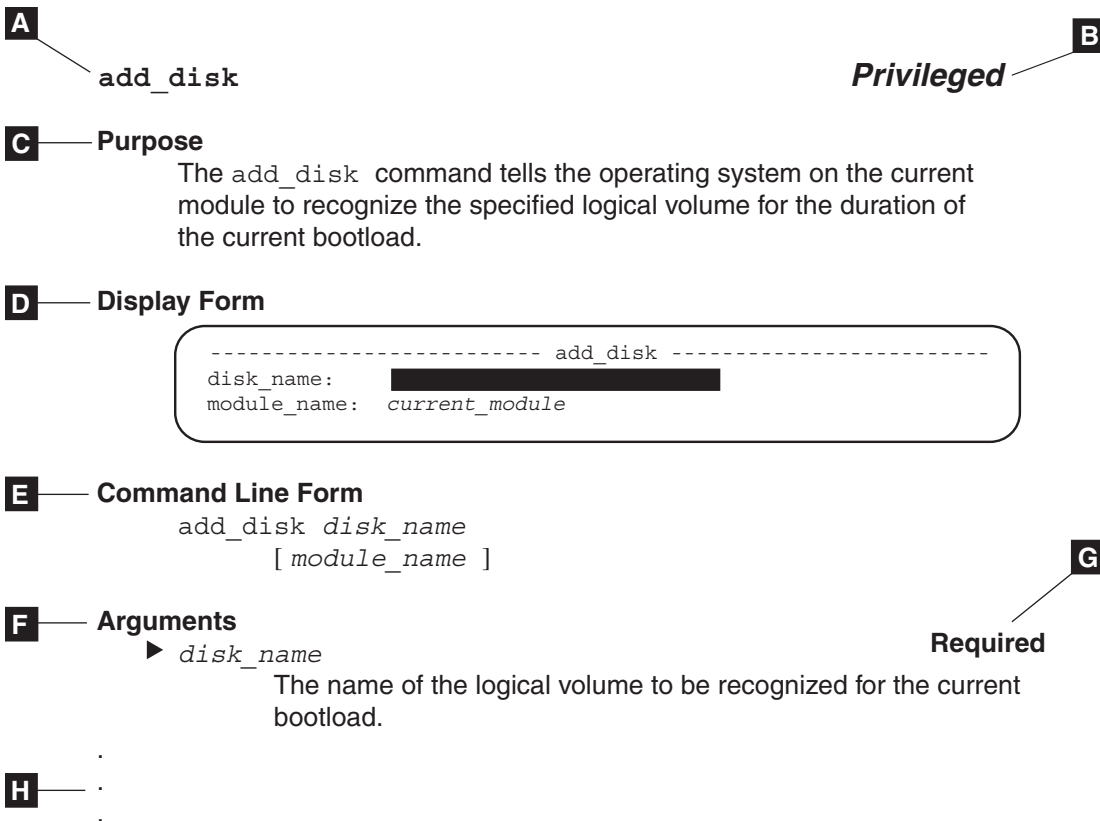
```
%dev#m1>system>acl>system_default
```

```
w *.*
```

Format for Commands and Requests

Stratus manuals use the following format conventions for documenting commands and requests. (A *request* is typically a command used within a subsystem, such as

analyze_system.) Note that the command and request descriptions do not necessarily include each of the following sections.



`add_disk`

Privileged

C Purpose

The `add_disk` command tells the operating system on the current module to recognize the specified logical volume for the duration of the current bootload.

D Display Form

```
----- add_disk -----
disk_name: 
module_name:  current_module
```

E Command Line Form

```
add_disk disk_name
      [ module_name ]
```

F Arguments

► `disk_name`

Required

The name of the logical volume to be recognized for the current bootload.

H

A name

The name of the command or request is at the top of the first page of the description.

B Privileged

This notation appears after the name of a command or request that can be issued only from a privileged process.

C Purpose

Explains briefly what the command or request does.



D Display Form

Shows the form that is displayed when you type the command or request name followed by `-form` or when you press the key that performs the `DISPLAY FORM` function. Each field in the form represents a command or request argument. If an

argument has a default value, that value is displayed in the form.

The following table explains the notation used in display forms.

The Notation Used in Display Forms

Notation	Meaning
	Required field with no default value.
	The cursor, which indicates the current position on the screen. For example, the cursor may be positioned on the first character of a value, as in a11.
<i>current_user</i> <i>current_module</i> <i>current_system</i> <i>current_disk</i>	The default value is the current user, module, system, or disk. The actual name is displayed in the display form of the command or request.

E Command-Line Form

Shows the syntax of the command or request with its arguments. You can display an online version of the command-line form of a command or request by typing the command or request name followed by `-usage`.

The following table explains the notation used in command-line forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Specifying multiple values is **not** the same as specifying a star name. When you specify multiple values, you must separate each value with a space.

The Notation Used in Command-Line Forms

Notation	Meaning
<code>argument_1</code>	Required argument.
<code>argument_1...</code>	Required argument for which you can specify multiple values.
$\left\{ \begin{array}{l} \text{argument}_1 \\ \text{argument}_2 \end{array} \right\}$	Set of arguments that are mutually exclusive; you must specify one of these arguments.
<code>[argument_1]</code>	Optional argument.
<code>[argument_1]...</code>	Optional argument for which you can specify multiple values.
$\left[\begin{array}{l} \text{argument}_1 \\ \text{argument}_2 \end{array} \right]$	Set of optional arguments that are mutually exclusive; you can specify only one of these arguments.
Note: Dots, brackets, and braces are not literal characters; you should not type them. Any list or set of arguments can contain more than two elements. Brackets and braces are sometimes nested.	

F Arguments

Describes the command or request arguments. The following table explains the notation used in argument descriptions.

G The Notation Used in Argument Descriptions

Notation	Meaning
<code>CYCLE</code>	This argument has predefined values. In the display form, you display these values in sequence by pressing the key that performs the <code>CYCLE</code> function.
Required	<p>You cannot issue the command or request without specifying a value for this argument.</p> <p>If an argument is required but has a default value, it is not labeled Required since you do not need to specify it in the command-line form. However, in the display form, a required field must have a value—either the displayed default value or a value that you specify.</p>
(Privileged)	Only a privileged process can specify a value for this argument.

- H** The following additional headings may appear in the command or request description: Explanation, Error Messages, Examples, and Related Information.

Explanation

Explains how to use the command or request and provides supplementary information.

Error Messages

Lists common error messages with a short explanation.

Examples

Illustrates uses of the command or request.

Related Information

Refers you to related information (in this manual or other manuals), including descriptions of commands, subroutines, and requests that you can use with or in place of this command or request.

Online Documentation

The OpenVOS StrataDOC Web site is an online-documentation service provided by Stratus. It enables Stratus customers to view, search, download, print, and comment on OpenVOS technical manuals via a common Web browser. It also provides the latest updates and corrections available for the OpenVOS document set.

You can access the OpenVOS StrataDOC Web site, at no charge, at <http://stratadoc.stratus.com>. A copy of OpenVOS StrataDOC on supported media is included with this release. You can also order additional copies from Stratus.

For information about ordering OpenVOS StrataDOC on supported media, see the next section, “Ordering Manuals.”

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN™), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.
- Contact the Stratus Customer Assistance Center (CAC), using either of the following methods:
 - From the ActiveService Manager (ASM) web site, log on to your ASM account. Click the `Manage Issues` button, enter your site ID in the `Create New Issue For Site` box, and then click the pencil icon to the right of the box. Fill out the forms and select `Update`.

- Customers in North America can call the CAC at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see <http://www.stratus.com/go/support/ftserver/location> for Stratus CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

Commenting on This Manual

You can comment on this manual using one of the following methods. When you submit a comment, be sure to provide the manual's name and part number, a description of the problem, and the location in the manual where the affected text appears.

- From StrataDOC, click the `site feedback` link at the bottom of any page. In the pop-up window, answer the questions and click `Submit`.
- From any email client, send email to `comments@stratus.com`.
- From the ASM web site, log on to your ASM account and create a new issue as described in the preceding section, "Ordering Manuals."
- From an OpenVOS window, specify the command `comment_on_manual`. To use the `comment_on_manual` command, your system must be connected to the RSN. This command is documented in the manual *OpenVOS System Administration: Administering and Customizing a System* (R281) and the *OpenVOS Commands Reference Manual* (R098). You can use this command to send your comments, as follows.
 - If your comments are brief, type `comment_on_manual`, press `[Enter]` or `[Return]`, and complete the data-entry form that appears on your screen. When you have completed the form, press `[Enter]`.
 - If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press `[Enter]` or `[Return]`. Enter this manual's part number, R648, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press `[Enter]`.

NOTE

If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail request` of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Chapter 1

Introduction to Dynamic Linking

OpenVOS Release 17.1.0 and later and GNU Tools for OpenVOS, Release 3.5 and later, support dynamic linking of shared libraries. This chapter provides an introduction to the topics in the following sections:

- [“Shared Libraries” on page 1-1](#)
- [“Static and Dynamic Linking” on page 1-3](#)
- [“Dynamic Dependencies” on page 1-3](#)
- [“OpenVOS and UNIX” on page 1-4](#)

Shared Libraries

Shared library functionality supports dynamically-linked OpenVOS user applications. Such applications can consist of a main program module (with the `.pm` suffix) and one or more shared libraries. This section contains the following topics:

- [“Description of a Shared Library” on page 1-1](#)
- [“Shared Library File Type” on page 1-2](#)
- [“Considerations” on page 1-2](#)

Description of a Shared Library

Instead of having just one large executable program module, a dynamically-linked application can consist of a smaller main program module and one or more shared libraries. At runtime, the *dynamic linker* brings a program and its shared libraries together. A *shared library* is an executable file that requires a main program module; it cannot be executed by itself.

Shared libraries defer some program binding tasks until runtime. The same shared library can be used by different programs. Shared libraries can be loaded into an already-running program in one of the following ways:

- When a program loads, all of the shared libraries on which it depends also get loaded at the same time. In this case, the binder only incorporates instructions to

load the shared libraries into the program. The actual contents of the libraries are not incorporated into the program until load-time.

- Shared libraries can be explicitly loaded by the program after it starts. In this case, the program calls the `dlopen` function to load shared libraries. The program can unload explicitly-loaded shared libraries by calling the `dlclose` function.

See the *OpenVOS POSIX.1 Reference Guide* (R502) for more information about the `dlopen` and `dlclose` functions.

Shared Library File Type

OpenVOS supports a new type of file: the shared library. Like relocatable object modules (`*.obj`, `*.o`), they are binder input. Shared libraries have some of the characteristics of other file types:

- Like program modules (`*.pm`), they are loadable binder output.
- Like program modules, they contain code and data; the code and read-only data are the same for all users; and other data are unique and private to each process.
- Like relocatable object modules, shared libraries contain definitions of and references to global symbols, which the dynamic linker resolves.
- Like relocatable object modules, there are locations that have to be relocated or modified at runtime once the shared library's load address is known.

Although shared library file names normally end in `.so`, this is not required. When a programmer uses the binder's `-pm_name` argument to name a shared library, that name is used verbatim. Likewise, when the dynamic linker searches for a shared library, it uses the name as specified, with no assumed suffix. The only place where a `.so` suffix is assumed is when the binder is constructing a default name for a shared library, or when the binder is searching for libraries specified with the `-lname` option.

Considerations

Shared libraries cannot share data between processes. When two processes use the same shared library, they only share code and read-only data. Each process has its own copy of the library's static and external data.

When you update a shared library, you must ensure that the new library is compatible with the application. If a shared library is replaced with a newer but incompatible version, applications may exhibit unexpected behavior. Also, if a shared library is replaced with an earlier version of itself, applications that depend on the later version could fail.

Static and Dynamic Linking

A *static program* is complete in that it contains all user code and runtimes. All compile, bind, and link operations must be completed before you can run the program. In order to change a program's functionality or to modify a library that the program uses, you must rebind or recompile a statically-linked program.

A *dynamically-linked program* is incomplete in that some of its parts reside in one or more shared libraries. Dynamic linking allows you to bind multiple files into an executable program at runtime. This is useful because it allows parts of a program to be modified or upgraded without having to rebind the entire application. Dynamically linked programs have the following benefits that are not available with statically-linked applications:

- You can install a bug fix or an upgrade to a system library without having to rebind applications.
- You can maintain and deploy different versions of the same shared library without having to rebind. For example, you can have production versions of shared libraries that are fully optimized with little or no debugging information and fully debuggable versions of the same shared libraries.
- You can extend the capabilities of a program at runtime. Applications can load additional libraries at runtime and enhance their own functionality.

Dynamic Dependencies

Unlike static applications, dynamically-linked applications have dependencies on shared libraries. An application may dynamically link in one or more shared libraries. Some of these shared libraries may, in turn, dynamically link in other shared libraries. An application can depend on an entire tree of shared libraries, where the libraries on one level have dependencies on other shared libraries in the level below. [Figure 1-1](#) shows a shared library dependency tree. At runtime, an application can run successfully only if it can find all of the shared libraries on which it and all of its shared libraries depend.

The dynamic linker traces down through the shared library dependency tree and loads one instance of each library. If, as shown in [Figure 1-1](#), an application has dependencies on three shared libraries (`libone.so`, `libtwo.so`, and `libthree.so`), and both `libone.so` and `libtwo.so` depend on `libfour.so`, the dynamic linker loads `libfour.so` only once.

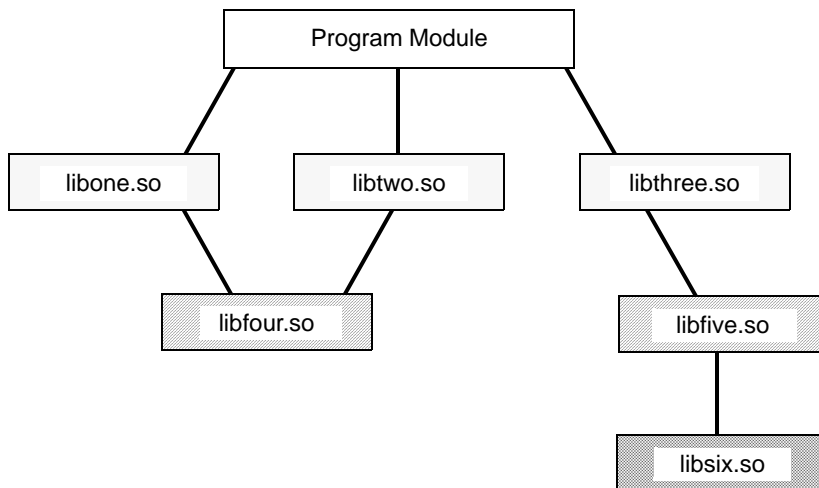


Figure 1-1. Shared Library Dependency Tree

OpenVOS and UNIX

The OpenVOS implementation of shared libraries supports development features found in both the OpenVOS and UNIX environments. The OpenVOS implementation of shared libraries and dynamic linking is compatible with many UNIX implementations. If you have used shared libraries and dynamic linking on UNIX, you will find many familiar features in the OpenVOS implementation of them.

The binder and dynamic linker can search for shared libraries using OpenVOS library paths. They can also search for shared libraries using the UNIX `LD_LIBRARY_PATH` environment variable. Both mechanisms are implemented so that full OpenVOS functionality is maintained while also providing support for porting applications developed in a UNIX environment.

The OpenVOS dynamic linker accepts both POSIX-style and OpenVOS-style file names and path names. Path names may contain the substitution sequence `$ORIGIN` and `${ORIGIN}` and may begin with the `(master_disk)` OpenVOS command function. As a result, the slash `/` character is always interpreted as a path name delimiter, not as a file name character, and the opening and closing parenthesis `()`, dollar-sign `$`, and opening and closing brace `{ }` characters are reserved.

The entire OpenVOS GNU Tools command set and the OpenVOS `vcc` compiler also accept POSIX-style and OpenVOS-style file names and path names. The OpenVOS binder, when invoked by `gcc`, `g++`, or `vcc`, also accepts POSIX-style file names and path names on its command line. See the *OpenVOS Commands Reference Manual* (R098) for more information about the `vcc` and `bind` commands.

Chapter 2

Creating Shared Libraries and Dynamically-Linked Applications

You can create shared libraries and dynamically-linked applications using OpenVOS Standard C, GNU C, or GNU C++ tools. This chapter contains the following sections:

- [“Creating Dynamically-Linked Applications with OpenVOS Standard C” on page 2-1](#)
- [“Creating Dynamically-Linked Applications with GNU C or C++” on page 2-4](#)
- [“Creating Applications with Other Languages” on page 2-5](#)

Creating Dynamically-Linked Applications with OpenVOS Standard C

You can use the OpenVOS `cc` command to compile application code. You can then use the `bind` command to bind the compiled code into program modules or shared libraries. Alternatively, you can use just the `vcc` command to both compile and bind your application. This section covers the following topics:

- [“Using the `cc` Command to Compile Applications” on page 2-1](#)
- [“Using the `bind` Command to Bind Applications” on page 2-2](#)
- [“Using the `vcc` Command to Compile and Bind Applications” on page 2-3](#)

Using the `cc` Command to Compile Applications

You compile code for shared libraries in the same way that you compile code for static applications. On other platforms, code in shared libraries is typically compiled to produce *position-independent code* (PIC), while code in the main program module is typically compiled to produce non-PIC. OpenVOS allows both PIC and non-PIC code in shared libraries and main program modules. Compared to a PIC version of the same shared library, a non-PIC shared library may take slightly longer to load but will probably run slightly faster. In either case, the performance differences are very small.

The `cc` compiler produces PIC by default, so you do not need to set any compiler options to place your code into shared libraries. See the *OpenVOS Standard C User's Guide* (R364) for information about how to use the `cc` command.

Using the `bind` Command to Bind Applications

Binding a shared library is essentially the same as binding any other program module. Input to the binder can be any of the following:

- OpenVOS object modules
- Executable and linking format (ELF) object modules as generated by `vcc`
- Archive (`.a`) files
- Other shared libraries

You can still use a binder control file, and you can examine the resulting output with `display_program_module` and other OpenVOS commands. The `bind` command has an argument and several command-line options specifically for building shared libraries. See the *OpenVOS Commands Reference Manual* (R098) for detailed information about the `bind` command.

The `-shared` Argument

The `bind` command's `-shared` argument specifies whether to generate a shared library or a program module. When this argument is specified, the binder generates a shared library (`name.so`, by default). Otherwise, it generates a program module (`name.pm`, by default).

Command-Line Options

The `bind` command has several command-line options that you can use to build shared libraries. See the *OpenVOS Commands Reference Manual* (R098) for descriptions of these options. The position of these options on the command line is significant. Changing the order in which options appear can produce unintended results.

When the binder is processing a `-lname` command-line option, it may find both an archive version (`*.a`) and a shared version (`*.so`) of a library. In such a case, the `-Bdynamic` option tells the binder to select the shared version or to select the archive version if the shared version is not available. The `-Bstatic` option tells the binder to select the archive version or to select the shared version if the archive version is not available. If neither option is specified, the binder looks for the archive version only and ignores the shared version. The following fragment of a `bind` command line illustrates this point:

```
bind myprog.obj -Bdynamic -lmylib -Bstatic -lmyotherlib ...
```

In this example, the binder selects `libmylib.so` over `libmylib.a` (assuming that both libraries exist), but selects `libmyotherlib.a` over `libmyotherlib.so`.

(assuming that both libraries exist). These files are input files. The resulting output file would be named `myprog.pm` because it is named after the first input file.

The position of the `-lname` option on the command line is also significant. Its position can impact whether or not the binder can resolve a reference to a symbol. When you use the option to name a library (`-lmylib`), the binder searches that library for unresolved symbol definitions at the point in the command line where it is given. Consider the following fragment of a `bind` command line:

```
bind objA.so objB.so -lmylib objC.so ...
```

In this example, the binder searches `libmylib` for all undefined symbols in objects placed **before** `-lmylib` on the command line (`objA.so`, `objB.so`). If another object (`objC.so`) that is placed after the `-lmylib` option contains a symbol that is defined only in `libmylib`, the reference to the symbol cannot be resolved.

Shared Library Names

Although it is common to use `-l` or `-lname` when binding libraries, it is not a requirement. You can use a shared or archive library name such as `libutil.so` as-is on the command line. This is the only way to reference shared or archive libraries in a binder control file.

Because the binder does not use a file's suffix to determine the type of an input file, a shared or archive library file name can have any suffix, even `.a`, `.o`, or `.obj`. The `.so` suffix does not necessarily indicate that a file is a shared library.

Symbol Resolution

When binding an application with shared libraries, the binder performs symbol resolution as it would when binding a static application, with the following differences:

- By default, the binder does not check for undefined symbols when binding a shared library. The shared library is an incomplete executable, and the binder does not know what other libraries will be linked in at run time. Specifying the `-zdefs` command-line option causes the binder to check for undefined symbols. It is good practice to use this option during application development.
- The binder does not issue a warning for multiply-defined symbols. For example, if symbol `x` is defined in both the main program and a shared library, the binder will use the symbol definition from the main program.

Using the `vcc` Command to Compile and Bind Applications

Typically, programmers use a wrapper command such as `vcc` (or `gcc/g++`) to both compile and bind dynamically-linked applications. The `vcc` command compiles the application code and passes binder arguments and command-line options to the `bind` command. See the *OpenVOS Commands Reference Manual* (R098) for a detailed description of the `vcc` command.

To make it easier to port applications developed in UNIX environments to OpenVOS, the `vcc` command accepts the same binder options and uses the same semantics as those used in UNIX environments. Consider the following example:

```
vcc -o sub.so -shared sub.c
vcc -o main.pm main.c (current_dir)>sub.so
```

In the preceding example, the first line compiles the source module (`sub.c`) and binds it into a shared library. The second line compiles the source module (`main.c`) and binds it with the shared library (`sub.so`).

Use the `vcc` command's `-Wl` option to specify binder arguments and options to pass to `bind`. Separate multiple arguments and options with commas (no spaces) as shown in the following example:

```
vcc -Wl,-Bdynamic,-soname=libx.so.1 ....
```

The `vcc` command-line options are the same as the similarly named `gcc` and `g++` options with the following exceptions:

- `-Bdynamic` and `-Bstatic`—For compatibility with prior versions, by default, the `vcc` command produces statically-linked programs. Use the `-dynamic`, or `-Wl, -Bdynamic` `vcc` options to specify dynamic linking.
- `-fpic` and `-fPIC`—These are the same as in `gcc` and `g++`. However, `vcc` generates position-independent code by default, regardless of whether you specify one of these options.

Creating Dynamically-Linked Applications with GNU C or C++

Like `vcc`, the `gcc` and `g++` commands are wrapper commands that compile application code and pass binder arguments and options to the `bind` command. See the *GNU Tools for OpenVOS: User's Guide* (R453) for information about the `gcc` and `g++` commands.

When creating shared libraries with the `gcc` and `g++` commands, keep in mind the default settings for the following options:

- `-static`—By default, programs created with `gcc` and `g++` are now dynamically linked. To link programs statically, you must specify the `-static` binder option.

NOTE

The default was `-static` in OpenVOS GNU Tools Release 3.4 and earlier.

- `-fpic` and `-fPIC`—By default, `gcc` and `g++` generate position-dependent code. To create a shared library with position-independent code, you must specify the `-fpic` or `-fPIC` option (they are equivalent on V Series modules).

Creating Applications with Other Languages

You can create shared libraries and dynamically-linked programs from source code written in other languages supported on OpenVOS. The development environment is similar to that described in “[Using the `cc` Command to Compile Applications](#)” on [page 2-1](#) and “[Using the `bind` Command to Bind Applications](#)” on [page 2-2](#).

Chapter 3

Running and Debugging Dynamically-Linked Applications

This chapter provides the following information about running and debugging dynamically-linked applications:

- [“Running Dynamically-Linked Applications” on page 3-1](#)
- [“Debugging Dynamically-Linked Applications” on page 3-6](#)

Statically-linked and dynamically-linked versions of the same program should behave identically. As on any operating system platform, a dynamically-linked program exhibits a slight performance degradation when it starts but virtually none after that. If you want a program to start as quickly as possible, link it statically.

Running Dynamically-Linked Applications

Running a dynamically-linked program is no different from running a statically-linked program. Simply invoke the dynamically-linked main program module and the application runs. The dynamic linker automatically takes care of finding and loading the shared libraries.

NOTE _____

You receive an error if you attempt to run a dynamically-linked program on a module running OpenVOS Release 17.0.x or earlier.

This section discusses the following topics:

- [“Understanding the Dynamic Linker” on page 3-2](#)
- [“The Application Runtime Environment” on page 3-5](#)

Understanding the Dynamic Linker

The dynamic linker is active whenever a dynamically-linked program starts up or when a `dl*` function is called (see [“Debugging Dynamically-Linked Applications” on page 3-6](#)). A part of the dynamic linker also gets control at program termination in order to handle shared-library termination code.

When an application starts, the dynamic linker loads all of the shared libraries with which the main program module was bound, along with all shared libraries on which those shared libraries depend. Similarly, a call to the `dlopen` function loads the specified shared library and all libraries on which it depends. The dynamic linker runtimes are thread-safe. Multiple threads may call `dl*` functions, and the dynamic linker will handle them sequentially.

The dynamic linker performs five steps when loading a shared library. The following sections describe these steps:

- [“Searching For and Opening Shared Library Files” on page 3-2](#)
- [“Loading Shared Libraries into Memory” on page 3-2](#)
- [“Resolving Symbol References” on page 3-4](#)
- [“Relocating Shared Library Addresses” on page 3-4](#)
- [“Handling Initialization/Termination Code” on page 3-4](#)

Searching For and Opening Shared Library Files

The dynamic linker searches for and opens all of the shared library files on which the application depends. This includes all of the shared libraries required by the program module, as well as all libraries that those libraries require, and so on. See [“Dynamic Dependencies” on page 1-3](#) for a full description of these dependencies.

The dynamic linker uses files exactly as they are specified; it does not assume any special suffixes for shared library names.

Loading Shared Libraries into Memory

After the dynamic linker finds each shared library file, it loads it into memory. It loads shared libraries in *breadth-first* order: that is, first, it loads all libraries required by the main program module, then it loads all libraries that those libraries require, and so on. This process continues until all required libraries have been loaded. If a required library has already been loaded, it is not loaded again. [Figure 3-1](#) illustrates this concept.

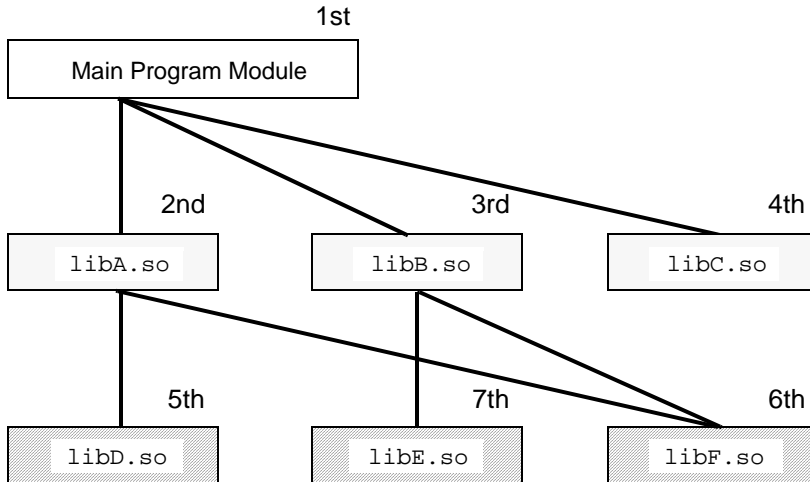


Figure 3-1. Shared Library Breadth-First Load Order

In the example shown in [Figure 3-1](#), the load order is as follows:

1. Main Program Module
2. libA.so
3. libB.so
4. libC.so
5. libD.so
6. libF.so
7. libE.so

NOTE

The dynamic linker loads libF.so before libE.so because libA.so depends on it. It loads libF.so only once, even though both libA.so and libB.so depend on it.

The dynamic linker determines shared libraries' memory requirements. For each shared library, it allocates the required memory from the user heap and reads all of the shared library's regions into it. It assigns permissions to the regions as follows:

- read and execute permissions to code regions
- read permission to symtab and maps regions
- read and write permissions to all other regions

Resolving Symbol References

After loading shared libraries, the dynamic linker resolves symbol references in the main program module and shared libraries. It uses the following rules, in the order in which they are shown, to look for symbol definitions:

1. If the symbol reference is in a shared library that was bound with the `-Bsymbolic` option, check the symbol definitions in this shared library.
2. Check the symbol definitions in the main program module.
3. Check the symbol definitions in all shared libraries that were loaded when the program started, in the order in which they were loaded.
4. Check the OpenVOS kernel.

The dynamic linker takes the first symbol definition it finds, and all subsequent definitions for that symbol are ignored. It does not issue any warnings for multiply-defined symbols. If no definition is found, the dynamic linker either displays a warning (when the program starts) or returns an error (on a `dlopen` call).

Relocating Shared Library Addresses

After resolving global symbol references, the dynamic linker relocates the addresses of the shared libraries. *Relocation* adjusts all absolute address references to reflect the actual load point of the shared library, instead of the tentative load point assigned to it by the binder.

Handling Initialization/Termination Code

After the dynamic linker relocates shared-library addresses, it handles initialization and termination code. Program modules and shared libraries can contain initialization code that runs when they are loaded as well as termination code that runs when a library is unloaded or the process terminates. The primary use of initialization and termination code is to execute constructors and destructors of C++ objects with global or static storage duration.

Initialization code is invoked immediately before control is passed to `main()`. Termination code is normally invoked when the program terminates, but in some circumstances (for example, if the program terminates abnormally), it is never invoked. Initialization and termination code is executed in the following order:

1. Initialization code is run in reverse-load order: the initialization of the last shared library is run first, and the initialization code of the main program module is run last.
2. Termination code is run in load order: the main program's termination code is run first, and the termination code of the last shared library loaded is run last.

NOTE

Thread-local static and global variables, if they are initialized, can be initialized only to constant values. They cannot use static constructors and destructors.

The Application Runtime Environment

Although running a dynamically-linked application is as simple as invoking its main program module, its runtime environment must be set up properly for it to run successfully. At runtime, the dynamic linker needs to find all of the shared libraries on which the program module depends. If it cannot locate a shared library, it generates a runtime error that identifies the missing library file.

When you deploy an application, keep in mind that its runtime environment is often different from the development runtime environment. Ensure that the application's runtime environment contains all of the shared libraries it requires and that they are located where the dynamic linker can find them. If a problem arises, you can adjust the runtime object library paths, provide a missing shared library file, or set the `LD_LIBRARY_PATH` environment variable. See [“The LD_DEBUG Environment Variable” on page 3-7](#) for more information.

You can use the `list_dynamic_dependencies` and `ldd` commands to check that the runtime environment of a dynamically-linked application is set up properly. These commands list all of the shared libraries on which the specified program module or shared library depends, and they also flag shared library files that cannot be found.

The `list_dynamic_dependencies` command works like a traditional OpenVOS command, and `ldd` is compatible with the `ldd` command on UNIX systems. See the *OpenVOS Commands Reference Manual* (R098) for information about the `list_dynamic_dependencies` and `ldd` commands.

Debugging Dynamically-Linked Applications

You can use the OpenVOS `debug` command and the OpenVOS GNU Tools `gdb` command to debug dynamically-linked applications. This section discusses the following topics:

- [“Using `debug`” on page 3-6](#)
- [“Using `gdb`” on page 3-6](#)
- [“The `LD_DEBUG` Environment Variable” on page 3-7](#)

Using `debug`

Use the OpenVOS `debug` command to troubleshoot dynamically-linked programs in the same way that you would debug a static program. You cannot examine or set breakpoints in a shared library that has not yet been loaded. Since OpenVOS loads a program module and all of its dependent shared libraries before the initial `debug` prompt, you can examine and set breakpoints in them before the program starts running.

You cannot examine or set breakpoints in shared libraries loaded with `dlopen` until that `dlopen` call is complete. For example, if your program is bound with `liba.so` and `libb.so`, and it also calls `dlopen` to open `libc.so` and `libd.so`, you can examine and set breakpoints in the main program, `liba.so`, and `libb.so` at the initial `debug` prompt. However, you cannot debug `libc.so` and `libd.so` until the `dlopen` call that loads them has been executed.

Using `gdb`

The `gdb` debugger differs from the OpenVOS debugger in that the initial command prompt is presented before the main program or its dependent shared libraries have been loaded. At this point, `gdb` allows you to examine and set breakpoints in the main program but not in the shared libraries. To avoid this problem, set a temporary breakpoint in the main function, then run the program. By the time `gdb` hits this breakpoint, the dependent shared libraries will have been loaded and so can be examined.

As with the OpenVOS debugger, `gdb` does not allow you to examine or debug shared libraries loaded with `dlopen` until the `dlopen` call that loads them has run.

The LD_DEBUG Environment Variable

You can use the LD_DEBUG environment variable to help debug problems with dynamic linking. If you set LD_DEBUG, the dynamic linker generates diagnostic messages as it runs. The LD_DEBUG environment variable has the following options:

- `files`: Report file opening and closing
- `initfini`: Report constructor/destructor invocation
- `libs`: Report on shared library searches
- `reloc`: Report on progress of relocations
- `symbols`: Report symbol lookups
- `help`: Help with LD_DEBUG

You can combine multiple arguments. For example:

```
LD_DEBUG=files,libs
```

To see a list of options, set LD_DEBUG=help and run a dynamically-linked program. See the *OpenVOS POSIX.1 Reference Guide* (R502) for more information about using environment variables.

Chapter 4

The `dl*` Functions

The POSIX functions `dlopen`, `dlclose`, `dlsym`, `dladdr`, and `dlerror` allow a program to load and access shared libraries during program execution. This chapter discusses these functions in the following sections:

- “[The `dlopen` Function and Shared Library Groups](#)” on page 4-1
- “[The `dlclose` Function](#)” on page 4-3
- “[The `dlsym` Function](#)” on page 4-3
- “[The `dladdr` Function](#)” on page 4-3
- “[The `dlerror` Function](#)” on page 4-3

See the *OpenVOS POSIX.1 Reference Guide* (R502) for more information about these functions.

The `dlopen` Function and Shared Library Groups

The `dlopen` function loads new shared libraries into a running program. Each call to `dlopen` creates a new shared library group consisting of the specified shared library and all of its dependencies. A *shared library group* is an ordered set of shared libraries used for symbol lookups. The main program module and all of the shared libraries on which it depends constitute the *initial group*.

Shared library groups prevent symbol definitions in one group from preempting definitions of the same symbols in another group. This means that two shared libraries opened by calls to `dlopen` can define and reference the same symbol name without interfering with each other.

Groups can be local or global in scope. Symbols defined inside a local group are only visible to that group. Symbols in global groups are visible to all subsequently-loaded groups (local or global). The main program module and its dependencies are global by default. Other groups are local by default.

When you specify a shared library with `dlopen`, the same loading and processing steps apply as with any other shared library:

- Search for and open shared library files (the specified file and the files on which it depends).

- Load these shared libraries into memory (if one is already loaded, it is not reloaded).
- Resolve symbol references.

Symbol references in all libraries, including those loaded by `dlopen`, are resolved according to the following rules:

1. If the symbol reference is in a shared library that was bound with the `-Bsymbolic` option, check the symbol definitions in this shared library.
 2. Check the symbol definitions in the main program module.
 3. Check the symbol definitions in all of the shared libraries that were loaded when the program started, in the order in which they were loaded.
 4. Check the symbol definitions in all previously-opened global shared library groups, in the order in which they were defined. Within each group, the shared libraries are visited in breadth-first dependency order.
 5. Check the shared library group defined by this `dlopen` call. Within the group, the shared libraries are visited in dependency order.
 6. If the shared library (in which the reference is being made) was initially loaded as part of a non-global shared library group, check each shared library in that group in breadth-first dependency order.
 7. Check the system kernel entry points. The shared library's own `references_kernel` flag controls which kernel symbols are visible to it.
- Relocate shared library addresses.
 - Handle initialization and termination code.

If you set the `dlopen` `RTLD_GLOBAL` flag, symbol definitions in the shared library group defined by this call to `dlopen` become visible to all groups that are created with subsequent calls to `dlopen`. Unlike local groups, you cannot unload these global groups by calling `dlclose`.

A single shared library may belong to multiple groups. The shared library named in a `dlopen` call is the *group founder*, regardless of whether that shared library was previously loaded or not.

For example, consider a program that makes two calls to `dlopen` to open two shared libraries, `libfirst.so` and `libsecond.so`, each of which defines the function `cleanup`. In this case, the two shared libraries are in separate groups, and each will see its own definition of `cleanup`; the definition in `libfirst.so` does not preempt the definition in `libsecond.so`.

The `dlclose` Function

The `dlclose` function unloads a shared library that was loaded by a call to `dlopen`. A shared library may be unloaded only when:

- It is not a member of any global shared library group
- Its open count (that is, the number of times `dlopen` has opened it, minus the number of times `dlclose` has closed it) is zero
- No other currently-loaded shared library depends on it

The `dlsym` Function

The `dlsym` function obtains the address of an external symbol name in a shared library group. The symbol must be available to be defined in the program. If you are looking for a symbol in the main program module, `dlsym` can find it only if the binder exported it. You can force the binder to output all symbols by using the `--export-dynamic` option when binding the main program module.

The `dladdr` Function

The `dladdr` function takes a function pointer and attempts to return that function's name and path name information. Applications often use this information to generate diagnostic information when an error occurs (for example, when the code finds a memory exception).

The `dlerror` Function

The `dlerror` function returns a pointer to a null-terminated character string that describes the last error that occurred during a call to `dlclose`, `dlopen`, `dlsym`, or `dladdr`. You can use this information to generate diagnostic output when the dynamic linker returns an error to an application.

Chapter 5

Shared Library Advanced Topics

This chapter discusses advanced topics for shared libraries. It contains the following sections:

- [“Binder Shared Library Search Rules” on page 5-1](#)
- [“Dynamic Linker Runtime Search Rules” on page 5-5](#)
- [“The Binder `-soname` Option and File-Level Versioning” on page 5-5](#)
- [“Binding Shared Libraries” on page 5-6](#)
- [“The Binder `--as-needed` Option” on page 5-7](#)
- [“Symbol Visibility in Shared Libraries” on page 5-8](#)
- [“Shared System Libraries” on page 5-10](#)
- [“The `LD_PRELOAD` Environment Variable” on page 5-11](#)

Binder Shared Library Search Rules

The OpenVOS `bind` command employs several library-search mechanisms. Some of these mechanisms have always been features of the binder. Others, used to process shared libraries and create dynamically-linked programs, first became available in OpenVOS Release 17.1.0.

Some of these search mechanisms are familiar to OpenVOS application programmers. Others are commonly used on UNIX platforms. In providing compatibility with both legacy OpenVOS and UNIX shared-library programming environments, the binder utilizes a complex set of library search rules when looking for symbol definitions.

To find library files, the binder searches through a hierarchy of locations and processes a variety of different types of library files (shared, static, and archive). When the binder first starts, it creates an internal list of the OpenVOS object library paths. Later, the binder uses this ordered list to search for certain types of libraries. It creates this internal list from the following sources:

- Directories named with the `-search` argument on the `bind` command line
- Directories named with the `search:` directive in the binder control file

- Directories listed in the object library path list of the process

Explicitly Named Input File Searches

If a binder control file is specified, the binder first looks for the input files explicitly named by the `modules:` directive in the binder control file. Object files specified as simple file names are searched for using the binder's internal list of OpenVOS object library paths.

Next, the binder looks in the input files explicitly named on the `bind` command line. It processes these files in the order in which they appear on the command line. Absolute object file path names specified on the command line are used exactly as they are specified. A relative path name is resolved relative to the current directory. A simple file name must reside in the current directory. The binder does not look for them elsewhere.

Archive File Searches

The `bind` command supports archive libraries as input files. You create archive files with the `ar` command, which is part of GNU Tools for OpenVOS. The binder recognizes that an input file is an archive library by examining its contents, not by its file name suffix (even though archive files typically have a `.a` suffix).

The binder searches through the archive file to see if it defines any symbols that are currently undefined. If it finds such a symbol definition, it loads the defining object module from that archive. Since that object module can contain new undefined symbols, the binder continues to search and load from the archive until it cannot find any more object modules to load. The binder performs this process only once, at the point that it sees the archive file. It is not read again.

Searches Specified by the `-lname` Option

You can specify libraries with the `-lname` option of the `bind` command. Because the libraries specified by the `-lname` option are explicitly named input files, the binder processes them (along with other input files) in the order in which they appear on the command line.

The `-lname` option search mechanism is similar to the one in UNIX environments. The binder creates two file names from the `-l` option (`libname.so` and `libname.a`). It then searches through directories specified by `-search` on the command line, the `search:` directive in a binder control file, and finally through the OpenVOS object library paths (in that order) and looks for a library named `libname.so` or `libname.a`. It uses the first `libname.*` library it finds. If it finds a shared library named `libname.so` first, it will use that library, even if `-Bstatic` is set. The `-Bdynamic` and `-Bstatic` options only determine which file to use in the case of a search tie (which can occur if both `libname.so` and `libname.a` reside in same directory). It is common practice to keep shared and static versions of the same library in the same object library directory.

To illustrate the search rules, suppose that your object library search paths include the `first_object_library` directory and the `second_object_library` directory, in that order.

The `first_object_library` directory contains the following libraries:

```
libA.so
libB.a
libC.so
libD.a
libE.a
libE.so
```

The `second_object_library` directory contains the following libraries:

```
libC.a
libD.so
libE.so
```

Given the preceding environment, a library search selects libraries as shown in the following examples:

```
bind -lA
    Selects libA.so in first_object_library, regardless of whether
    -Bstatic or -Bdynamic is set.

bind -lB
    Selects libB.a in first_object_library, regardless of whether
    -Bstatic or -Bdynamic is set.

bind -lC
    Selects libC.so in first_object_library, regardless of whether
    -Bstatic or -Bdynamic is set.

bind -lD
    Selects libD.a in first_object_library, regardless of whether
    -Bstatic or -Bdynamic is set.

bind -Bstatic -lE
    Selects libE.a in first_object_library.

bind -Bdynamic -lE
    Selects libE.so in first_object_library.
```

Searches Done After Bind File and Command-Line Processing

After the binder control file and command-line processing is complete, the following searches are performed in the order listed:

- [“Default Shared Library Searches” on page 5-4](#)
- [“Implicit Shared Library Searches” on page 5-4](#)
- [“OpenVOS Object Library Searches” on page 5-5](#)

Default Shared Library Searches

If `-Bdynamic` is in effect after command-line processing is complete, the binder searches through its internal list of OpenVOS object library paths for files and links that have a `.dso` file name suffix and that contain a required symbol definition. If it finds such a file, the binder dynamically links it into the program. The binder does not perform this search if the `no_library` or `-Bstatic` option is set.

Each directory in the object library path may contain zero, one, or many `*.dso` files and links. If a directory contains multiple `*.dso` files, the application cannot assume that they will be loaded in any particular order. For example, if `libA.dso` must be loaded ahead of `libB.dso` because it preempts some of the latter’s symbols, `libA.dso` must be in a directory that is searched before the directory that `libB.dso` is in.

If the binder encounters a `*.dso` shared library that has already been dynamically linked, the binder does not search or link it again.

Implicit Shared Library Searches

After the binder performs its default shared-library searches, it searches for implicit shared libraries. These are shared libraries on which previously-seen shared libraries depend but that the binder has not yet encountered. The binder does not use its internal list of object library directories to search for implicit shared libraries. Instead, it uses the dynamic linker runtime search rules described in [“Dynamic Linker Runtime Search Rules” on page 5-5](#). The binder does not perform this search if the `no_library` or `-znodels` option is set.

The binder is performing a practice run at this point. It checks to make sure that all required symbols are defined, common symbols all have the right sizes, and so on. It does not actually load or attempt to run any shared libraries. It not only has to read the named shared library (see the `libY.so` example in [“Binding Shared Libraries” on page 5-6](#)), but it also must find (on its own) all of the libraries on which this one depends (see the `libZ.so` example in [“Binding Shared Libraries” on page 5-6](#)). The binder uses different rules—identical to the runtime dynamic linker search rules—for this search.

OpenVOS Object Library Searches

After the binder performs an implicit shared-library search, it searches through the traditional OpenVOS object libraries in the following order:

1. All directories specified on the command line with the `-search` argument
2. All directories specified in the binder control file with the `search:` directive
3. All directories listed in the object library paths

The binder does not perform this search if the `no_library` option is set.

Dynamic Linker Runtime Search Rules

The dynamic linker uses the following rules (in the order in which they are presented) to locate and open all of the shared library files on which a main program module or a shared library depends:

1. If the shared library is specified as an absolute or relative path name, the binder uses that path name as given. Relative path names are relative to the current directory.
2. The binder looks in each directory specified by the `LD_LIBRARY_PATH` environment variable. The value of this variable is a colon-separated list of directory names.

NOTE

For historical compatibility with UNIX, a semicolon may be used in place of one colon in this list.

3. If the program module or shared library that referenced this shared library was bound with the `-rpath` option, the binder looks in the directories specified by that option.
4. The binder looks in the directories listed in the current object library path (except for `(current_directory)`).
5. The binder looks in `(master_disk)>lib` (OpenVOS-style) or `/lib` (POSIX-style).
6. Finally, the binder looks in `(master_disk)>usr>lib` (OpenVOS-style) or `/usr/lib` (POSIX-style).

The Binder `-soname` Option and File-Level Versioning

Typically, you bind an application with the latest version of a shared library. You can use file-level versioning if you want the application to always dynamically link to the same version of that shared library.

File-level versioning allows you to make changes to a shared library that is incompatible with current applications, without causing those current applications to fail.

To implement file-level versioning for dynamically-linked programs, use symbolic links and the binder's `-soname` option as follows:

1. Create the latest version of a shared library (`libutil.6.so`) using the `-soname` option. This option specifies that at runtime, an application should dynamically link in `libutil.6.so`.
2. Create a symbolic link (`libutil.so`) that points to the latest version of the shared library (`libutil.6.so`).
3. Bind `libutil.so` into the application. At runtime, the application dynamically links in `libutil.6.so` but not `libutil.so`.
4. When `libutil.7.so` (again, bound with the `-soname` option) is released, change the `libutil.so` link to point to `libutil.7.so`.
5. Bind `libutil.so` into the updated version of the application.

At runtime, the newly-built application dynamically links in the newest version (`libutil.7.so`), but applications previously bound against the link to version 6 will continue to dynamically link in version 6.

Binding Shared Libraries

You can bind a shared library into a main program module or into another shared library. When the binder reads a shared library as an input file, it does the following:

- Adds the shared library's symbols (defined and undefined) to its symbol table
- Records the shared library as a dependency of the output file (program module or shared library)
- Notes all of the shared libraries that the input file depends on, for use later in the bind process

When binding a program that uses shared libraries, the binder creates lists of the defined and undefined symbols that it finds in the program modules and in all of the shared libraries on which the program module depends. With this list, the binder can choose which static objects to load from the object libraries, and it can create a list of undefined symbols.

To build this list, the binder notes all of the shared-library dependencies in the entire bound-in shared-library dependency tree (see [Figure 1-1](#)). This includes looking in the bound-in shared libraries themselves, all of the second-level shared libraries they depend on, all of the third-level shared libraries that the second-level libraries depend on, and so on. Essentially, the binder practices the dynamic-linking operation, using the

dynamic linker's runtime library search rules, instead of its own search rules. See [“Dynamic Linker Runtime Search Rules” on page 5-5](#) for details. This practice run occurs after the binder reads all of the input files but before it searches for *.obj files to satisfy undefined symbols. The following example illustrates this concept:

```
bind -shared -pm_name libZ.so ...
bind -shared -pm_name libY.so ... libZ.so
bind -pm_name progX.pm ... libY.so
```

In this example, `progX.pm` depends on `libY.so`, which, in turn, depends on `libZ.so`. In order to check for undefined symbols when binding `progX.pm`, the binder needs to locate and read `libZ.so`, not just `libY.so`. To locate `libZ.so`, the binder uses the dynamic linker's **runtime** search rules, not its own search rules.

The Binder *--as-needed* Option

When you bind a shared library into a program module or another shared library, it is not actually copied into the program module or other shared library. Instead, a note is made in the program module or shared library to load the other shared library at runtime.

Sometimes, a shared library is bound in even if it is not needed; that is, the program runs the same with or without the shared library. In this case, loading that shared library is unnecessary runtime overhead.

To avoid this overhead, bind the shared library in *as-needed* mode. The *as-needed* mode instructs the binder to check whether or not a shared library satisfies some outstanding external symbol reference, and load the shared library only if it does.

As-needed mode is controlled by a pair of binder command-line options. The *--as-needed* option turns on as-needed mode, and the *--no-as-needed* (the default) option turns it off. Each option affects only shared libraries that follow it on the command line.

For example, consider the following `bind` command for a simple program that consists of one object module and one shared library:

```
bind main.o libutil.so
```

Suppose that `main.o` calls the `sort()` function, which is defined in `libutil.so`. When the program is run, `libutil.so` is loaded and the link is made from `main.o` to `sort()` in `libutil.so`.

In the following example, the `bind` command uses *as-needed* mode with the same object module and shared library:

```
bind main.o --as-needed libutil.so --no-as-needed
```

It produces a program identical to the program produced by the first example.

However, if you change `main.o` so that it no longer calls `sort()`, the two examples produce different results. In the first example, `libutil.so` is loaded at runtime, even though it is not needed. However, in the second example, `libutil.so` is not loaded at runtime.

Symbol Visibility in Shared Libraries

Just like a single compilation unit, a single shared library or dynamically-linked program can have symbols that are local to itself or visible to other dynamically-linked objects. This is known as *symbol visibility*.

Symbol definitions in dynamically-linked programs can be hidden or global. A *hidden* symbol is visible only within a single shared library. A *global* symbol is visible to the entire program.

By default, symbols are visible to other dynamically-linked objects. You control symbol visibility with command-line and binder control file options.

A symbol's visibility can affect when links to it are resolved. If a symbol in a shared library has hidden visibility, another definition cannot preempt it, which means that references to it from elsewhere in that shared library can be resolved at bind time. On the other hand, if a symbol in a shared library has global visibility, resolution has to be deferred until runtime. You can optimize the performance of dynamic applications by minimizing the number of global symbols in a shared library.

In addition to global and hidden visibility, a shared library symbol can have symbolic visibility. This is a hybrid of global and hidden visibility. A symbol with *symbolic visibility* is visible to other shared objects, but references to it from within the shared library in which it is defined are resolved at bind time.



CAUTION

As with the binder's `-Bsymbolic` option, misuse of the `symbolic` option could result in two versions of one symbol being used.

Controlling Visibility from `gcc` and `g++`

To set visibility from within `gcc/g++` programs, include code similar to the following:

```
void __attribute__((visibility ("protected")))
f () { /* Do something. */; }
int i __attribute__((visibility ("hidden")));
```

See *Using the GNU Compiler Collection (GCC)* (R549m) for more information about using the `visibility` attribute.

Controlling Visibility in the Binder Control File

The binder control file `visibility:` directive provides symbol-by-symbol control of visibility. It has the following syntax:

```
visibility: visibility-specifier[, visibility-specifier...];
```

Where *visibility-specifier* is of the form:

```
symbol-name[*] { default | global |  
                  protected | symbolic |  
                  hidden | internal }
```

The `default` and `global` values give a symbol visibility to all shared libraries, and they defer resolution of that symbol until runtime. The `protected` and `symbolic` values are the same as `global` visibility, except that references to the symbol from within the defining shared library are resolved at bind time. The `hidden` and `internal` values make a symbol local to the defining shared library, and all references to it are resolved at bind time.

As with other binder control file directives, the symbol name may have a final asterisk, making it a star name that affects all symbols matching that star name. If more than one *visibility-specifier* applies to a single symbol, the last specifier given applies and overrides any prior directives. This allows you to give an entire class of symbols (for example, `$_c_* hidden`) one visibility, while giving certain of those symbols (for example, `$_c_set_errno symbolic`) a different visibility. Binder control file-specified visibility also overrides any visibility specified in the source code.

The following example sets `symbol1` to be hidden and all symbols starting with `symbol2` to be protected:

```
visibility:  
    symbol1 hidden;  
    symbol2* protected;
```

The `-version-script` Binder Option

Version-script files on Linux and Solaris set the version number and visibility of individual symbols. For compatibility, the OpenVOS binder can process version scripts written for these platforms. However, on OpenVOS, a version script can only control symbol visibility (symbol versioning is currently not supported). You should remove all `[version-name]` definitions from version-script files on OpenVOS.

The version-script file contains `global:` and `local:` labels that specify the visibility of the subsequent symbols (*symbolname*) in the definition. An asterix character specifies all defined global symbols whose visibility has not been explicitly set. This is normally used to make all symbols local to a single shared library by default.

In general, the syntax for a version-script file is:

```
{
    [ label : ] symbolname [ , symbolname... ] ;
    .
    .
    .
};
```

The following example shows a typical version-script file:

```
{
    global:
    mainsymbol,
    supportname;
    local:
    *;
};
```

This version script specifies that two symbols, `mainsymbol` and `supportname`, should have global visibility, and everything else should have local (hidden) visibility.

Shared System Libraries

In order for applications using shared libraries to work correctly (especially shared libraries loaded with `dlopen`), OpenVOS provides the following shared versions of the system libraries:

```
>system>object_library
>system>c_object_library
>system>posix_object_library
>system>posix_object_library>pthread
```

If you want to automatically bind shared versions of the system libraries into your program, make sure that dynamic mode (`-Bdynamic`) is in effect at the end of command-line processing.

The binder uses the default shared library (`*.dso`) mechanism to automatically bind in shared system libraries. See [“Default Shared Library Searches” on page 5-4](#) for more information about default shared libraries.

The binder finds the shared system libraries in the same directories in which the static objects (`*.obj`) are located. Since the code for a particular function is functionally the same in both the static and dynamic libraries, you will get the same runtimes regardless of whether you are binding statically or dynamically.

When building a thread-aware program, you can bind in the shared system `pthread` library in one of two ways.

- You can use the `-l` option just as you would in a UNIX environment, as shown in the following example:

```
gcc main.o -lpthread
```

- Alternatively, you can bind in the shared system `pthread` library by ensuring that it is in the OpenVOS object-library search path.

At runtime, the dynamic linker loads the shared system libraries from the `(master_disk)>lib` directory.

The `LD_PRELOAD` Environment Variable

The `LD_PRELOAD` variable specifies a whitespace-separated list of shared libraries to be loaded at program startup. These libraries are loaded after the main program module and before any other shared libraries. Typically, you use this environment variable to introduce substitute versions of system functions. These substitute versions can access the original functions by calling `dlsym` with a value of `RTLD_NEXT` for the *handle* argument.

Appendix A

Troubleshooting Dynamically-Linked Applications

This appendix discusses some common problems that can occur when developing and deploying dynamically-linked applications. It contains the following sections:

- [“Dynamic Linker Cannot Find a Shared Library” on page A-1](#)
- [“Undefined Symbol Error When Library Is Opened by `dlopen`” on page A-1](#)

Dynamic Linker Cannot Find a Shared Library

Sometimes the dynamic linker may not be able to find a shared library. This problem can occur because the missing shared library resides in the current directory. By default, the current directory is not on the dynamic linker’s shared-library search list.

You can work around this problem by setting the library path to include the current directory as follows:

```
LD_LIBRARY_PATH=.
```

Alternatively, you can bind the missing shared library as a path name instead of a simple file name (`>libN.so` instead of `libN.so`).

Undefined Symbol Error When Library Is Opened by `dlopen`

Sometimes an undefined symbol error occurs when an application opens a shared library by calling the `dlopen` function. This problem can occur because the symbol is defined in the main program module. The binder may have omitted resolving this symbol reference because it only exports the symbols it thinks need to be exported from the main program module.

You can solve this problem by explicitly exporting all global symbols. Use the `--export-dynamic` option when binding the main program module.

Alternatively, you can put the required symbol definitions into a shared library that is already being bound into the program.

Index

A

Archive files, 2-2

--as-needed
 bind command-line option, 5-7

B

-Bdynamic
 bind command-line option, 2-2
bind command, 2-2
 archive library input file, 5-2
 --as-needed option, 5-7
 -Bdynamic option, 2-4
 -Bstatic option, 2-4
 command-line options, 2-2
 -fPIC option, 2-4
 -fpic option, 2-4
 library search rules, 5-1
 after processing, 5-4
 explicitly named input files, 5-2
 -lname option, 5-2
 --no-as-needed option, 5-7
 -rpath option, 5-5
 -version-script option, 5-9
 -zdefs option, 2-3

Binding shared libraries, 5-6

-Bstatic
 bind command-line option, 2-2

C

cc command, 2-1

Commands

 bind, 2-2
 cc, 2-1
 debug, 3-6
 gcc/g++, 1-4, 2-4
 gdb, 3-6
 ldd, 3-5
 list_dynamic_dependencies, 3-5
 vcc, 1-4, 2-3

D

debug command, 3-6
Dependencies, dynamic, 1-3
 breadth-first order, 3-2, 4-2
dladdr function, 4-3
dlclose function, 4-3
dlopen function, 4-1
dlsym function, 4-3
Dynamic
 linker, 1-1
 assign permissions, 3-4
 described, 3-2
 file names, 1-4
 initialization code, 3-4
 path names, 1-4
 resolve symbol references, 3-4
 runtime search rules, 5-5
 termination code, 3-4
 linking, 1-3
 programs, 1-3
 benefits of, 1-3
 running, 3-1
 runtime environment, 3-5

E

Environment variables

 LD_DEBUG, 3-7
 LD_LIBRARY_PATH, 1-4, 3-5, 5-5

F

File-level versioning, 5-5

-fPIC
 gcc/g++ command option, 2-5
 default values, 2-5
 vcc command option, 2-4
 default values, 2-4

-fpic

- gcc/g++ command option, 2-5
 - default values, 2-5
- vcc command option, 2-4
 - default values, 2-4

Functions

- dladdr, 4-3
- dlclose, 4-3
- dLError, 4-3
- dlopen, 4-1
- dlsym, 4-3

G

- g++ command, 1-4
 - options, 2-4
- gcc command, 1-4
 - options, 2-4
- gdb command, 3-6

I

- Implicit shared library searches, 5-4
- Initialization code, 3-5

L

- LD_DEBUG environment variable, 3-7
- LD_LIBRARY_PATH environment variable, 1-4,
3-5, 5-5
- ldd command, 3-5
- Library searches
 - default shared, 5-4
 - implicit shared, 5-4
 - OpenVOS object, 5-5
 - rules, 5-1
- list_dynamic_dependencies
command, 3-5

N

- no-as-needed
 - bind command-line option, 5-7

O

OpenVOS

- object library searches, 5-5
- shared system libraries, 5-10

P

- PIC, 2-1
- Position-independent code (PIC). See PIC
- Programs
 - dynamic, 1-3
 - static, 1-3

R

Runtime

- environment, 3-5
- search rules, 5-5

S

- Shared libraries, 1-1
 - binding, 5-6
 - dependencies, 1-3, 5-6
 - file name suffix, 1-2
 - file type, 1-2
 - loading, 1-2
 - names, 2-3
 - OpenVOS system libraries, 5-10
 - symbol visibility, 5-8
- Static
 - linking, 1-3
 - program, 1-3
- Symbol
 - references, resolving, 3-4
 - resolution, 2-3
 - visibility in shared libraries, 5-8

T

- Termination code, 3-5

V

- vcc command, 1-4, 2-3
 - options, 2-4
- Versioning, file-level, 5-5
- Version-script files, 5-9