HP-UX Linker and Libraries User's Guide

HP 9000 Computers

B2355-90655

November 1997

Legal Notices

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

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Corporate Offices:

Hewlett-Packard Co.
3000 Hanover St.
Palo Alto, CA 94304

Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Use of this manual and flexible disc(s), compact disc(s), or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Preface

This *Guide* covers the following topics:

- Chapter 1, “What's New in Recent Releases,” lists new features added in recent releases.
- Chapter 2, “What Happens When You Compile and Link a Program,” provides details on compiling and linking programs.
- Chapter 3, “Linker Tasks,” lists many ways you can specify how you want your program linked.
- Chapter 4, “Linker Tools,” list the tools available in the linker toolset.
- Chapter 5, “Creating and Using Libraries,” discusses all aspects of both archive and shared libraries.
- Chapter 6, “Shared Library Management Routines,” explains how to explicitly load libraries at run time using shared library management routines.
- Chapter 7, “Position-Independent Code,” describes how to write position-independent assembly code.
- Chapter 8, “Ways to Improve Performance,” discusses several ways to optimize your program.
- Appendix A, “Using Mapfiles,” describes mapfiles.
- Glossary contains definitions of important terms in this manual.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. The manual part number changes when extensive technical changes are incorporated.

- November 1997, Edition 1, part number B2355-90655. This manual supersedes *HP-UX Linker and Libraries User's Guide* part number B2355-90655. The main reason for this new edition is to document new functionality for the HP-UX 11.00 release:

- Add the +ee linker option to export symbols.
- Add 64-bit linker toolset support for linker options.
- Add 64-bit mode linker tools and describe the enhancements to the 32-bit mode toolset.
- Describe 32-bit and 64-bit mode behavior differences.
- Add 64-bit mode dynamic path searching mechanisms.
- Add 64-bit mode symbol binding semantics.
- Add the dl* shared library management routines for 64-bit mode support and describe enhancement to the shl_load routines for shared libraries.
- Add init/fini style initializers for 64-bit mode support for shared libraries.
- Add the BIND_BREADTH_FIRST flag to the shl_load routine to control search path behavior.
- Add description of support for ELF object file format.

- April 1997, Edition 1, part number B2355-90654. This manual supersedes *Programming on HP-UX* part number B2355-90652. The main reason for this new edition is to document new functionality for the HP-UX 10.30 release:
  - Announce linker thread-safe features for ld, dld.sl, crt0.o, and libdld.sl.
  - Add the +pd *size* linker option to set the virtual memory page size for program data.
  - Add the +pi *size* linker option to set the virtual memory page size for program instructions.
  - Add the +k linker option (see *ld*(1)) to only create an executable if no errors are found at link time.
  - Add the chatr +k option to enable or disable kernel-assisted branch prediction.
  - Add the chatr +pd *size* and +pi *size* virtual memory page setting options.

- July 1996, Edition 1, part number B2355-90653. This manual supersedes *Programming on HP-UX* part number B2355-90652. The main reason for this new edition is to document new functionality for the HP-UX 10.20 release and to describe what's ahead in a future release of the linker toolset:
  - Add a `-B symbolic` option to help improve shared library performance.
  - Introduce the `fastbind` tool to improve the start up time of programs that use shared libraries.
  - Introduce the *Linker and Libraries Online User Guide*.
  - Announce changes in PA-RISC hardware compatibility—PA-RISC 1.1 systems, by default, generate PA-RISC 1.1 code; PA-RISC 2.0 systems generate 2.0 code.
  - Describe compatibility warnings generated by the linker and dynamic loader for HP 9000 architecture issues and linker toolset features that may change in a future release.
  - Describe what's changing in a future release of the linker toolset.
  - Add the `+Ostaticprediction` option to use with profile-based optimization.
- January 1995, Edition 1, part number B2355-90652. This manual supersedes *Programming on HP-UX* part number B2355-90026. The main reason for this new edition is to document new functionality for the HP-UX 10.0 release:
  - Update path names to reflect the new System V Release 4 file system. Some of the changes are:
    - Most files in `/lib` are now in `/usr/lib`.
    - Most optional products are in `/opt`. For example, HP C is in `/opt/ansic`, HP C is in `/opt/CC`, HP FORTRAN/9000 is in `/opt/fortran`, and HP/DDE is in `/opt/langtools/dde`.
  - Caution against mixing shared and archive libraries.
  - Describe a new library-level versioning scheme for shared libraries.
  - Update the chapter on profile-based optimization.
  - Describe changes in optimization levels 3 and 4.

- Describe thread-safe interfaces `shl_get_r` and `shl_gethandle_r`.
- Add a new `BIND_TOGETHER` flag to the `shl_load` routine.
- Add a new chapter "Porting Applications to HP-UX."

1 What's New in Recent Releases

This section contains information about recent releases of the HP-UX linker toolset:

For This Release

The HP-UX 11.00 linker toolset contains new features:

If you use the 32-bit mode linker toolset, see the following items:

- “PA-RISC Changes in Hardware Compatibility” updated in this chapter.
- “Exporting Symbols with +ee” on page 81.
- “Changes in Future Releases” updated in this chapter.

If you use the 64-bit mode linker toolset, see the following items:

- “PA-RISC Changes in Hardware Compatibility” updated in this chapter.
- “64-bit Mode Linker Toolset Compatibility with De Facto Industry Standards” described in this chapter.
- “64-bit Mode ELF Object File Format” described in this chapter.
- “Dynamic Path Searching for Shared Libraries” on page 177 describes differences in the run time searching of shared libraries.
- “Shared Library Symbol Binding Semantics” on page 178 describes differences in shared library binding semantics.
- New 64-bit mode linker options, symbols, and features, described in“New Features for 64-bit Mode Linking” in this chapter.
- Unsupported 32-bit mode features, behavior, and linker options, described in “64-bit Mode Link-time Differences” and“64-bit Mode Run Time Differences” in this chapter.
- “64-bit Mode Initializers” on page 210 describes the init/fini support for 64-bit mode shared libraries.

- “The dlopen Shared Library Management Routines” on page 240 describes the `dl*` family of shared library management routines for 64-bit mode.
- “BIND_BREADTH_FIRST Modifier” on page 222 describes the flag added to the `shl_load` routine to modify search behavior.
- “Changes in Future Releases” updated in this chapter.

For Previous Releases

The following items were added in the HP-UX 10.30 release:

- “Linker Thread-Safe Features” on page 50.
- “Options to Improve TLB Hit Rates” on page 273.
- The `+k` linker option (see *ld*(1)) to remove an executable if the link fails.
- The `+k chatr` option (see *chatr*(1)) to improve branch prediction on PA-RISC 2.0.

The following items were added in the HP-UX 10.20 release:

- “Improving Shared Library Performance with -B symbolic” on page 60.
- “Improving Shared Library Start-Up Time with fastbind” on page 293.
- “Online Help for Linker and Libraries” described in this chapter.
- “PA-RISC Changes in Hardware Compatibility” described in this chapter.
- “Linker Compatibility Warnings” on page 99.
- “Dynamic Loader Compatibility Warnings” on page 256.
- The `+Ostaticprediction` linker option described in the *ld*(1) man page to use with profile-based optimization

PA-RISC Changes in Hardware Compatibility

The HP-UX 10.20 release introduced HP 9000 systems based on the PA-RISC 2.0 architecture. Also, beginning with that release, HP compilers by default generate executable code for the PA-RISC architecture of the machine on which you are compiling.

In previous releases, the compilers generated PA-RISC 1.0 code on all HP 9000 Series 800 servers and PA-RISC 1.1 code on Series 700 workstations. HP compilers now by default generate PA-RISC 1.1 code on 1.1 systems and 2.0 code on 2.0 systems.

Using the `+DAportable` compiler option provides compatibility of code between PA-RISC 1.1 and 2.0 systems. Note that the HP-UX 10.10 release is the last supported release for PA-RISC 1.0 systems, so code generated by the HP-UX 10.20 release of HP compilers is not supported on PA-RISC 1.0 systems.

NOTE The `+DA1.0` option will be obsolete in a future release. You can achieve better performance on PA-RISC 1.1 and 2.0 systems by not using this option.

PA-RISC 2.0 Compatibility

The instruction set on PA-RISC 2.0 is a superset of the instruction set on PA-RISC 1.1. As a result, code generated for PA-RISC 1.1 systems will run on PA-RISC 2.0 systems. However, code generated for PA-RISC 2.0 systems will *not* run on PA-RISC 1.1 or 1.0. The linker issues a hardware compatibility warning whenever it links in any PA-RISC 2.0 object files:

```
/usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file
(sum.o) was detected. The linked output may not run on PA 1.x
system.
```

If you try to run a PA-RISC 2.0 program on a 1.1 system, you'll see a message like:

```
$ a.out
ksh: ./a.out: Executable file incompatible with hardware
```

In this example, the `+DAportable` compiler option can be used to create code compatible for PA-RISC 1.1 and 2.0 systems.

PA-RISC Architectures and Their System Models

The HP 9000 PA-RISC (Precision Architecture Reduced Instruction Set Computing) Series 700/800 family of workstations and servers has evolved from three versions of PA-RISC:

PA-RISC 1.0	The original version of PA-RISC first introduced on Series 800 servers. The following Series are included: 840, 825, 835/SE, 845/SE, 850, 855, 860, 865, 870/x00, 822, 832, 842, 852, 890, 808, 815, 635, 645.
PA-RISC 1.1	The second version of PA-RISC first introduced on Series 700 workstations. Newer Series 800 systems also use this version of the architecture. The following Series are included: 700, 705, 710, 715, 720, 725, 730, 735, 750, 755, B132L, B160L, B132L+, B180L, C100, C110, J200, J210, J210XC, 742i, 742rt, 743i, 743rt, 745i, 747i, 748i, 8x7, D (except Dx70, Dx80), E, F, G, H, I, K (except Kx50, Kx60, Kx70), T500, T520.
PA-RISC 2.0	The newest version of PA-RISC. The following Series are included: C160, C180, C180XP, C200, C240, J280, J282, J2240, Dx70, Dx80, Kx50, Kx60, Kx70, T600, V2200.

For More Information

- See your compiler online help or documentation for details on the `+DA` option.
- See the file `/opt/langtools/lib/sched.models` for a complete list of model numbers and their architectures. Use the command `model` to determine the model number of your system.

64-bit Mode Linker Toolset Compatibility with De Facto Industry Standards

The 64-bit mode linker and dynamic loader provide linking and loading behaviors found widely across the Unix industry, considered, with the SVR4 standards, to define the de facto industry standards. The following 64-bit linker behavior is compliant with de facto industry standard:

- ELF object file format and *libelf*(3x) routines
- Dynamic path searching
- Library-level versioning
- `dl*` family of dynamic loading routines
- Breadth-first symbol searching

The HP-UX 11.00 release maintains certain behaviors to make transition from 32-bit to 64-bit mode easier:

- Creation of default run-time path environment variable (RPATH) if no `ld +b` is seen on the link line, to improve transition from the 32-bit mode linker.
- `ld +compat` option for compatibility with 32-bit linking and loading behavior.

64-bit Mode ELF Object File Format

Starting with HP-UX release 11.00, the 64-bit linker toolset supports the ELF (*executable and linking format*) object file format. The 64-bit linker toolset provides new tools to display and manipulate ELF files. The *libelf*(3x) library routines provide access to ELF files. The command *elfdump*(1) displays contents of an ELF file.

The following options instruct the compiler to generate 64-bit ELF object code.

Option	Compiler
+DA2.0W	C and aC++
+DD64	C

See the HP-UX Software Transition Toolkit (STK) at `http://www.software.hp.com/STK/` for more information on the structure of ELF object files.

New Features for 64-bit Mode Linking

This section introduces new features of the 64-bit linker for HP-UX release 11.00.

64-bit Mode Linker Options

The *ld*(1) command supports the following new options in 64-bit mode:

Option	Action
`-dynamic`	Forces the linker to create a shared executable. The linker looks for shared libraries first and then archived libraries. This option is on by default when you compile in 64-bit mode.
`-noshared`	Forces the linker to create a fully bound archive program.
`-k` *filename*	Allows you to control the mapping of input section in the object file to segments in the output file.
`+[no]allowunsats`	Instructs the linker how to report errors for output files with unsatisfied symbols.
`+compat`	Instruct the linker to use 32-bit mode linking and dynamic loading behaviors.
`+[no]forceload`	Enables/disables forced loading of all the object files from archive libraries.[a]
`+hideallsymbols`	Hides all symbols from being exported.[a]
`+nodefaultmap`	Instructs the linker not to load the default mapfile. See the `-k` option.[a]

Option	Action
+noenvvar	Instructs the dynamic loader not to look at the LD_LIBRARY_PATH and SHLIB_PATH environment variables at runtime.[a]
+std	Instructs the linker to use SVR4 compatible linking and loading behaviors. Default for 64-bit mode.[a]
+stripunwind	Instructs the linker not to output the unwind table.
+vtype *type*	Produces verbose output about selected link operations.[a]

a. The linker accepts but ignores this option in 32-bit mode. It creates an executable (a.out).

64-bit Mode Linker-defined Symbols

The 64-bit linker reserves the following symbol names:

Symbol	Definition
__SYSTEM_ID	Largest architecture revision level used by any compilation unit
_FPU_STATUS	Initial value of FPU status register
_end	Address of first byte following the end of the main program's data segment; identifies the beginning of the heap segment
__TLS_SIZE	Size of the Thread Local Storage segment required by the program
__text_start	Beginning of the text segment
_etext	End of the text segment
__data_start	Beginning of the data segment

Symbol	Definition
_edata	End of initialized data
__gp	Global pointer value
__init_start	Beginning of the .init section
__init_end	End of the .init section
__fini_start	Beginning of the .fini section
__fini_end	End of the .fini section
__unwind_start	Beginning of the unwind table
__unwind_end	End of the unwind table

NOTE The linker generates an error if a user application also defines these symbols.

64-bit Mode Link-time Differences

The 64-bit mode linker toolset does not support the following 32-bit mode features.

Option or Behavior	Description
`-A` *name*	Specifies incremental loading. 64-bit applications must use shared libraries instead.
`-C` *n*	Does parameter type checking. This option is unsupported.
`-S`	Generates an initial program loader header file. This option is unsupported.
`-T`	Save data and relocation information in temporary files to reduce virtual memory requirements during linking. This option is unsupported.
`-q`, `-Q`, `-n`	Generates an executable with file type `DEMAND_MAGIC`, `EXEC_MAGIC`, and `SHARE_MAGIC` respectively. These options have no effect and are ignored in 64-bit mode.
`-N`	Causes the data segment to be placed immediately after the text segment. This option is accepted but ignored in 64-bit mode. If this option is used because your application data segment is large, then the option is no longer needed in 64-bit mode. If this option is used because your program is used in an embedded system or other specialized application, consider using mapfile support with the `-k` option.
`+cg` *pathname*	Specifies *pathname* for compiling I-SOMs to SOMs. This option is unsupported.

Option or Behavior	Description
+dpv	Displays verbose messages regarding procedures which have been removed due to dead procedure elimination. Use the `-v` linker option instead.
Intra-library versioning	Specified by using the `HP_SHLIB_VERSION` pragma (C and aC++) or `SHLIB_VERSION` directive (Fortran90). In 32-bit mode, the linker lets you version your library by object files. 64-bit applications must use SVR4 library-level versioning instead.
Duplicate code and data symbols	Code and data cannot share the same namespace in 64-bit mode. You should rename the conflicting symbols.
All internal and undocumented linker options	These options are unsupported.

For more information, see the *HP-UX Linker and Libraries Online User Guide* (`ld +help`).

64-bit Mode Run Time Differences

Applications compiled and linked in 64-bit mode use a run-time dynamic loading model similar to other SVR4 systems. There are two main areas where program startup changes in 64-bit mode:

- Dynamic path searching for shared libraries.
- Symbol searching in dependent libraries.

It is recommended that you use the standard SVR4 linking option (`+std`), which is on by default when linking 64-bit applications. There may be circumstances while you transition, that you need 32-bit compatible linking behavior. The 64-bit linker provides the `+compat` option to force the linker to use 32-bit linking and dynamic loading behavior.

The following table summarizes the dynamic loader differences between 32-bit and 64-bit mode:

Linker and Loader Functions	32-bit Mode Behavior	64-bit Mode Behavior
`+s` and `+b` *path_list* ordering	Ordering is significant.	Ordering is insignificant by default. Use `+compat` to enforce ordering.

Linker and Loader Functions	32-bit Mode Behavior	64-bit Mode Behavior
Symbol searching in dependent libraries	Depth-first search order.	Breadth-first search order. Use `+compat` to enforce depth first ordering.
Run time path environment variables	No run time environment variables by default. If `+s` is specified, then `SHLIB_PATH` is available.	`LD_LIBRARY_PATH` and `SHLIB_PATH` are available. Use `+noenv` or `+compat` to turn off run-time path environment variables.
`+b` *path_list* and `-L` *directories* interaction	`-L` *directories* recorded as absolute paths in executables.	`-L` *directories* are not recorded in executables. Add all directories specified in `-L` to `+b` *path_list*.

For more information on transition issues, see *HP-UX 64-bit Porting and Transition Guide*.

Changes in Future Releases

The following changes are planned in future releases.

- Support of ELF 32 object file format

  A future release will support the ELF 32 object file format.

- Future of `ld +compat` option

  The `+compat` linker option and support of compatibility mode may be discontinued in a future release.

- Support of `shl_load` shared library management routines

  A future release may discontinue support of the `shl_load` family of shared library management routines.

Online Help for Linker and Libraries

The *Linker and Libraries Online User Guide* is available for HP 9000 Series 700 and 800 systems. The online help comes with HP C, HP C++, HP aC++, HP Fortran, HP Pascal, and HP Micro Focus COBOL/UX. Online help can be accessed from any X Window display device, or from the *charhelp*(1) character-mode help browser.

Accessing Help with ld +help

To access the *Linker and Libraries Online User Guide* from the `ld` command line:

```
ld +help
```

Accessing Help with the HP CDE Front Panel

To access the *Linker and Libraries Online User Guide* if your HP compiler is installed on your system:

1. Click on the? icon on the HP CDE front panel.
2. The "Welcome to Help Manager" menu appears. Click on the HP Linker icon.

Accessing Help with the dthelpview Command

If your HP compiler is installed on another system or you are not running HP CDE, enter the following command from the system where your compiler is installed:

```
/usr/dt/bin/dthelpview -h linker
```

NOTE To make it easier to access, add the path `/usr/dt/bin` to your `.profile` or `.login` file.

Accessing Help with the charhelp Command

To access the *Linker and Libraries Online User Guide* from a character-mode terminal or terminal emulator:

`/opt/langtools/bin/charhelp ld`
See *charhelp*(1) for details.

2 What Happens When You Compile and Link a Program

This chapter describes the process of compiling and linking a program.

- “Compiling Programs on HP-UX: An Example” provides an overview of compiling on HP-UX.
- “Looking “inside” a Compiler” describes the process of creating an executable file in more detail.
- “Linking Programs on HP-UX” describes how `ld` creates an executable file from one or more object files.
- “Linking with Libraries” describes conventions for using libraries with `ld`.
- “Running the Program” describes the process of loading and binding programs at run time.
- “Linker Thread-Safe Features”describes the thread-safe features.

Compiling Programs on HP-UX: An Example

To create an executable program, you compile a source file containing a main program. For example, to compile an ANSI C program named `sumnum.c`, shown below, use this command (`-Aa` says to compile in ANSI mode):

```
$ cc -Aa sumnum.c
```

The compiler displays status, warning, and error messages to standard error output (`stderr`). If no errors occur, the compiler creates an executable file named `a.out` in the current working directory. If your `PATH` environment variable includes the current working directory, you can run `a.out` as follows:

```
$ a.out
Enter a number: 4
Sum 1 to 4: 10
```

The process is essentially the same for all HP-UX compilers. For instance, to compile and run a similar FORTRAN program named `sumnum.f`:

```
$ f77 sumnum.f     Compile and link sumnum.f.
   ...             The compiler displays any messages here.
$ a.out            Run the program.
  ...              Output from the program is displayed here.
```

Program source can also be divided among separate files. For example, `sumnum.c` could be divided into two files: `main.c`, containing the main program, and `func.c`, containing the function `sum_n`. The command for compiling the two together is:

```
$ cc -Aa main.c func.c
main.c:
func.c:
```

Notice that `cc` displays the name of each source file it compiles. This way, if errors occur, you know where they occur.

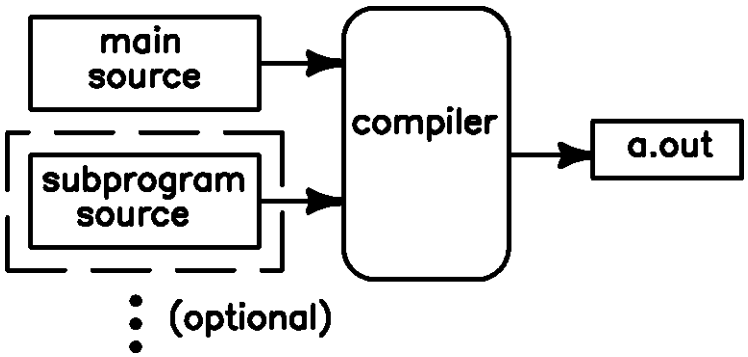
```
#include <stdio.h>              /* contains standard I/O defs */
int     sum_n( int n )           /* sum numbers from n to 1    */
{
  int   sum = 0;                 /* running total; initially 0 */
  for (; n >= 1; n--)            /* sum from n to 1            */
```

```
    sum += n;                    /* add n to sum               */
  return sum;                    /* return the value of sum    */
}

main()                           /* begin main program         */
{
  int   n;                       /* number to input from user  */
  printf("Enter a number: ");    /* prompt for number          */
  scanf("%d", &n);               /* read the number into n     */
  printf("Sum 1 to %d: %d\\n", n, sum_n(n)); /* display the sum */
}
```

Generally speaking, the compiler reads one or more source files, one of which contains a main program, and outputs an executable `a.out` file, as shown in “High-Level View of the Compiler”.

Figure 2-1 High-Level View of the Compiler



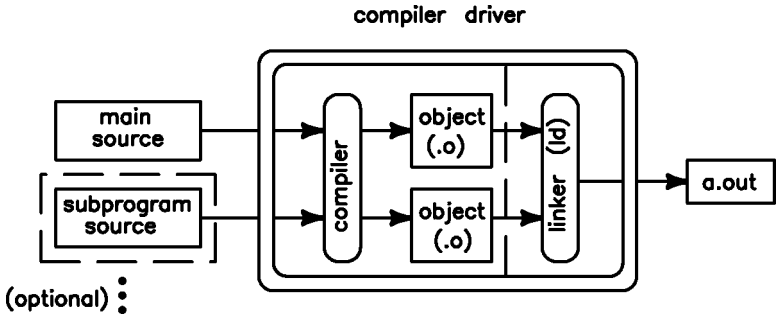
Looking "inside" a Compiler

On the surface, it appears as though an HP-UX compiler generates an `a.out` file by itself. Actually, an HP-UX compiler is a **driver** that calls other commands to create the `a.out` file. The driver performs different tasks (or **phases**) for different languages, but two phases are common to all languages:

1. For each source file, the driver calls the language compiler to create an object file. (See Also "What is an Object File?".)
2. Then, the driver calls the HP-UX linker (`ld`) which builds an `a.out` file from the object files. This is known as the **link-edit phase** of compilation. (See Also "Compiler-Linker Interaction".)

"Looking "inside" a Compiler" summarizes how a compiler driver works.

Figure 2-2 Looking "inside" a Compiler



The C, C++, FORTRAN, and Pascal compilers provide the `-v` (verbose) option to display the phases a compiler is performing. Compiling `main.c` and `func.c` with the `-v` option produced this output on a Series 700 workstation (\ at the end of a line indicates the line is continued to the next line):

```
$ cc -Aa -v main.c func.c -lm
cc: CCOPTS is not set.
main.c:
/opt/langtools/lbin/cpp.ansi main.c /var/tmp/ctmAAAa10102 \\
  -D__hp9000s700 -D__hp9000s800 -D__hppa -D__hpux \\
  -D__unix -D_PA_RISC1_1
cc: Entering Preprocessor.
/opt/ansic/lbin/ccom /var/tmp/ctmAAAa10102 main.o -O0 -Aa \\
  func.c:
/opt/langtools/lbin/cpp.ansi func.c /var/tmp/ctmAAAa10102 \\
  -D__hp9000s700 -D__hp9000s800 -D__hppa -D__hpux \\
```

```
  -D__unix -D_PA_RISC1_1
cc: Entering Preprocessor.
/opt/ansic/lbin/ccom /var/tmp/ctmAAAa10102 func.o -O0 -Aa
cc: LPATH is /usr/lib/pa1.1:/usr/lib:/opt/langtools/lib:
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -u main main.o func.o \\
  -lm -lc
cc: Entering Link editor.
```

This example shows that the `cc` driver calls the C preprocessor (`/opt/langtools/lbin/cpp.ansi`) for each source file, then calls the actual C compiler (`/opt/ansic/lbin/ccom`) to create the object files. Finally, the driver calls the linker (`/usr/ccs/bin/ld`) on the object files created by the compiler (`main.o` and `func.o`).

What is an Object File?

An **object file** is basically a file containing machine language instructions and data in a form that the linker can use to create an executable program. Each routine or data item defined in an object file has a corresponding **symbol name** by which it is referenced. A symbol generated for a routine or data definition can be either a **local definition** or **global definition**. Any reference to a symbol outside the object file is known as an **external reference**.

To keep track of where all the symbols and external references occur, an object file has a **symbol table**. The linker uses the symbol tables of all input object files to match up external references to global definitions.

Local Definitions

A local definition is a definition of a routine or data that is accessible only within the object file in which it is defined. Such a definition cannot be accessed from another object file. Local definitions are used primarily by debuggers, such as `adb`. More important for this discussion are global definitions and external references.

Global Definitions

A global definition is a definition of a procedure, function, or data item that can be accessed by code in another object file. For example, the C compiler generates global definitions for all variable and function definitions that are not `static`. The FORTRAN compiler generates global definitions for subroutines and common blocks. In Pascal, global definitions are generated for external procedures, external variables, and global data areas for each module.

External References

An external reference is an attempt by code in one object file to access a global definition in another object file. A compiler cannot resolve external references because it works on only one source file at a time. Therefore, the compiler simply places external references in an object file's symbol table; the matching of external references to global definitions is left to the linker or loader.

Compiler-Linker Interaction

As described in "Looking "inside" a Compiler", the compilers automatically call `ld` to create an executable file. To see how the compilers call `ld`, run the compiler with the `-v` (*v*erbose) option. For example, compiling a C program in 32-bit mode produces the output below:

```
$ cc -Aa -v main.c func.c -lm
cc: CCOPTS is not set.
main.c:
/opt/langtools/lbin/cpp.ansi main.c /var/tmp/ctmAAAa10102 \\
  -D__hp9000s700 -D__hp9000s800 -D__hppa -D__hpux \\
  -D__unix -D_PA_RISC1_1
cc: Entering Preprocessor.
/opt/ansic/lbin/ccom /var/tmp/ctmAAAa10102 main.o -O0 -Aa
func.c:
/opt/langtools/lbin/cpp.ansi func.c /var/tmp/ctmAAAa10102 \\
  -D__hp9000s700 -D__hp9000s800 -D__hppa -D__hpux \\
  -D__unix -D_PA_RISC1_1
cc: Entering Preprocessor.
/opt/ansic/lbin/ccom /var/tmp/ctmAAAa10102 func.o -O0 -Aa
cc: LPATH is /usr/lib/pa1.1:/usr/lib:/opt/langtools/lib:
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -u main main.o
func.o -lm -lc
cc: Entering Link editor.
```

The next-to-last line in the above example is the command line the compiler used to invoke the 32-bit mode linker, `/usr/ccs/bin/ld`. In this command, `ld` combines a **startup file** (`crt0.o`) and the two object files created by the compiler (`main.o` and `func.o`). Also, `ld` searches the `libm` and `libc` libraries.

In 64-bit mode, the startup functions are handled by the dynamic loader, `dld.sl`. In most cases, the `ld` command line does not include `crt0.o`.

NOTE

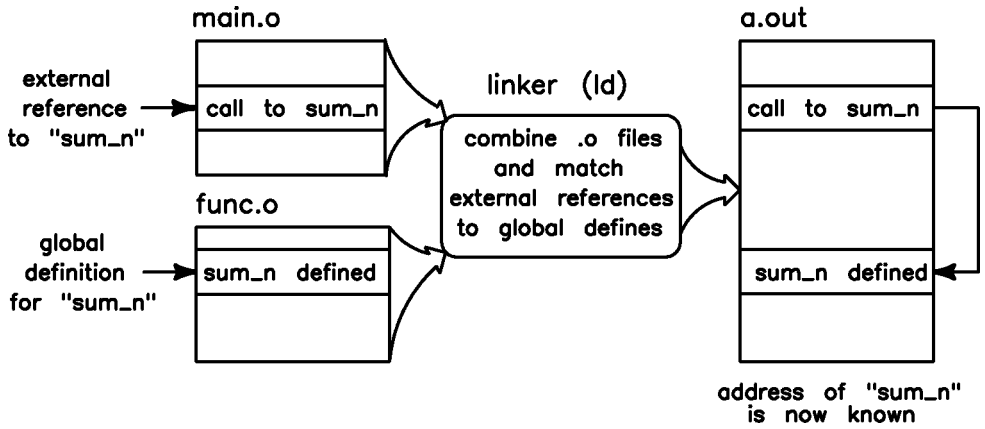
If you are linking any C++ object files to create an executable or a shared library, you must use the `CC` command to link. This ensures that `c++patch` executes and chains together your nonlocal static constructors and destructors. If you use `ld`, the library or executable may not work correctly and you may not get any error messages. For more information see the *HP C++ Programmer's Guide*.

Linking Programs on HP-UX

The HP-UX linker, `ld`, produces a single executable file from one or more input object files and libraries. In doing so, it matches external references to global definitions contained in other object files or libraries. It revises code and data to reflect new addresses, a process known as **relocation**. If the input files contain debugger information, `ld` updates this information appropriately. The linker places the resulting executable code in a file named, by default, `a.out`.

In the C program example (see "Compiling Programs on HP-UX: An Example") `main.o` contains an external reference to `sum_n`, which has a global definition in `func.o`. `ld` matches the external reference to the global definition, allowing the main program code in `a.out` to access `sum_n` (see Figure 2-3).

Figure 2-3 Matching the External Reference to sum_n



If `ld` cannot match an external reference to a global definition, it displays a message to standard error output. If, for instance, you compile `main.c` *without* `func.c`, `ld` cannot match the external reference to `sum_n` and displays this output:

```
$ cc -Aa main.c
/usr/ccs/bin/ld: Unsatisfied symbols:
   sum_n (code)
```

The crt0.o Startup File

Notice in the example in "Compiler-Linker Interaction" that the first object file on the linker command line is `/opt/langtools/lib/crt0.o`, even though this file was not specified on the compiler command line. This file, known as a **startup file**, contains the program's **entry point** that is, the location at which the program starts running after HP-UX loads it into memory to begin execution. The startup code does such things as retrieving command line arguments into the program at run time, and activating the dynamic loader (*dld.sl*(5)) to load any required shared libraries. In the C language, it also calls the routine `_start` in `libc` which, in turn, calls the main program as a function.

The 64-bit linker uses the startup file, `/opt/langtools/lib/pa_64/crt0.o`, when:

- The linker is in compatibility mode (`+compat`).
- The linker is in default standard mode (`+std`) with the `-noshared` option.

If the `-p` profiling option is specified on the 32-bit mode compile line, the compilers link with mcrt0.o instead of crt0.o. If the -G profiling option is specified, the compilers link with gcrt0.o. In 64-bit mode with the `-p` option, the linker adds `-lprof` before the `-lc` option. With the `-G` option, the linker adds `-lgprof`.

If the linker option `-I` is specified to create an executable file with profile-based optimization, in 32-bit mode `icrt0.o` is used, and in 64-bit mode the linker inserts `/usr/ccs/lib/pa20_64/fdp_init.o`. If the linker options `-I` and `-b` are specified to create a shared library with profile-based optimization, in 32-bit mode `scrt0.o` is used, and in 64-bit mode, the linker inserts `/usr/ccs/lib/pa20_64/fdp_init_sl.o`. In 64-bit mode, the linker uses the single 64-bit `crt0.o` to support these option.

For details on startup files, see *crt0*(3).

The Program's Entry Point

In 32-bit mode and in 64-bit statically-bound (`-noshared`) executables, the entry point is the location at which execution begins in the `a.out` file. The entry point is defined by the symbol `$START$` in `crt0.o`.

In 64-bit mode for dynamically bound executables, the entry point, defined by the symbol `$START$` in the dynamic loader (`dld.sl`).

The a.out File

The information contained in the resulting `a.out` file depends on which architecture the file was created on and what options were used to link the program. In any case, an executable `a.out` file contains information that HP-UX needs when loading and running the file, for example: Is it a shared executable? Does it reference shared libraries? Is it demand-loadable? Where do the code (text), data, and bss (uninitialized data) segments reside in the file? For details on the format of this file, see *a.out*(4).

Magic Numbers

In 32-bit mode, the linker records a **magic number** with each executable program that determines how the program should be loaded. There are three possible values for an executable file's magic number:

`SHARE_MAGIC`	The program's text (code) can be shared by processes; its data cannot be shared. The first process to run the program loads the entire program into virtual memory. If the program is already loaded by another process, then a process shares the program text with the other process.
`DEMAND_MAGIC`	As with `SHARE_MAGIC` the program's text is shareable but its data is not. However, the program's text is loaded only as needed — that is, only as the pages are accessed. This can improve process startup time since the entire program does not need to be loaded; however, it can degrade performance throughout execution.
`EXEC_MAGIC`	Neither the program's text nor data is shareable. In other words, the program is an unshared executable. Usually, it is not desirable to create such unshared executables because they place greater demands on memory resources.

By default, the linker creates executables whose magic number is `SHARE_MAGIC`. The following shows which linker option to use to specifically set the magic number.

Table 2-1 32-bit Mode Magic Number Linker Options

To set the magic number to:	Use this option:
SHARE_MAGIC	-n
DEMAND_MAGIC	-q
EXEC_MAGIC	-N

An executable file's magic number can also be changed using the `chatr` command (see “Changing a Program's Attributes with chatr(1)” on page 104). However, `chatr` can only toggle between `SHARE_MAGIC` and `DEMAND_MAGIC`; it cannot be used to change from or to `EXEC_MAGIC`. This is because the file format of `SHARE_MAGIC` and `DEMAND_MAGIC` is exactly the same, while `EXEC_MAGIC` files have a different format. For details on magic numbers, refer to *magic*(4).

In 64-bit mode, the linker sets the magic number to the predefined type for ELF object files (`\177ELF`). The value of the `E_TYPE` in the ELF object file specifies how the file should be loaded.

File Permissions

If no linker errors occur, the linker gives the `a.out` file read/write/execute permissions to all users (owner, group, and other). If errors occurred, the linker gives read/write permissions to all users. Permissions are further modified if the **umask** is set (see *umask*(1)). For example, on a system with umask set to 022, a successful link produces an `a.out` file with read/write/execute permissions for the owner, and read/execute permissions for group and other:

```
$ umask
022
$ ls -l a.out
-rwxr-xr-x   1 michael  users      74440 Apr  4 14:38 a.out
```

Linking with Libraries

In addition to matching external references to global definitions in object files, `ld` matches external references to global definitions in libraries. A **library** is a file containing object code for subroutines and data that can be used by other programs. For example, the standard C library, `libc`, contains object code for functions that can be used by C, C++, FORTRAN, and Pascal programs to do input, output, and other standard operations.

Library Naming Conventions

By convention, library names have the form:

`lib`*`name.suffix`*

name	is a string of one or more characters that identifies the library.
suffix	is `.a` if the library is an archive library or `.sl` if the library is a shared library. *suffix* is a number, for example `.0`, `.1`, and so forth, if library-level versioning is being used.

Typically, library names are referred to without the suffix. For instance, the standard C library is referred to as `libc`.

Default Libraries

A compiler driver automatically specifies certain default libraries when it invokes `ld`. For example, `cc` automatically links in the standard library `libc`, as shown by the `-lc` option to `ld` in this example:

```
$ cc -Aa -v main.c func.c
    ...
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -u main main.o \
func.o -lc
cc: Entering Link editor.
```

Similarly, the Series 700/800 FORTRAN compiler automatically links with the `libcl` (C interface), `libisamstub` (ISAM file I/O), and `libc` libraries:

```
$ f77 -v sumnum.f
   ...
/usr/ccs/bin/ld -x /opt/langtools/lib/crt0.o \
 sumnum.o -lcl -lisamstub -lc
```

The Default Library Search Path

By default, `ld` searches for libraries in the directory `/usr/lib`. (If the `-p` or `-G` compiler profiling option is specified on the command line, the compiler directs the linker to also search `/usr/lib/libp`.) The default order can be overridden with the `LPATH` environment variable or the `-L` linker option. `LPATH` and `-L` are described in "Changing the Default Library Search Path with -L and LPATH" on page 57.

Link Order

The linker searches libraries in the order in which they are specified on the command line — the **link order**. Link order is important in that a library containing an external reference to another library must precede the library containing the definition. This is why `libc` is typically the last library specified on the linker command line: because the other libraries preceding it in the link order often contain references to `libc` routines and so must precede it.

NOTE

If multiple definitions of a symbol occur in the specified libraries, `ld` does *not* necessarily choose the first definition. It depends on whether the program is linked with archive libraries, shared libraries, or a combination of both. Depending on link order to resolve such library definition conflicts is risky because it relies on undocumented linker behavior that may change in future releases. (See Also "Caution When Mixing Shared and Archive Libraries" on page 164.)

Running the Program

An executable file is created after the program has been compiled and linked. The next step is to run or load the program.

Loading Programs: exec

When you run an executable file created by `ld`, the program is loaded into memory by the HP-UX program loader, `exec`. This routine is actually a system call and can be called by other programs to load a new program into the current process space. The `exec` function performs many tasks; some of the more important ones are:

- Determine how to load the executable file by looking at its magic number. (See Also “The a.out File”.)
- Determine where to begin execution of the program — that is, the **entry point** — usually in `crt0.o`. (See Also “The crt0.o Startup File”.)
- When the program uses shared libraries, the `crt0.o` startup code invokes the dynamic loader (`dld.sl`), which in turn attaches any required shared libraries. If immediate binding was specified at link time, then the libraries are bound immediately. If deferred binding was specified, then libraries are bound as they are referenced. (See Also “What are Shared Libraries?” on page 126.)For details on `exec`, see the *exec*(2) page in the *HP-UX Reference*.

Binding Routines to a Program

Since shared library routines and data are not actually contained in the `a.out` file, the dynamic loader must **attach** the routines and data to the program at run time. Attaching a shared library entails mapping the shared library code and data into the process's address space, relocating any pointers in the shared library data that depend on actual virtual addresses, allocating the **bss segment**, and **binding** routines and data in the shared library to the program.

The dynamic loader binds only those symbols that are reachable during the execution of the program. This is similar to how archive libraries are treated by the linker; namely, `ld` pulls in an object file from an archive library only if the object file is needed for program execution.

Deferred Binding is the Default

To accelerate program startup time, routines in a shared library are not bound until referenced. (Data items are always bound at program startup.) This **deferred binding** of shared library routines distributes the overhead of binding across the execution time of the program and is especially expedient for programs that contain many references that are not likely to be executed. In essence, deferred binding is similar to **demand-loading**.

Linker Thread-Safe Features

Beginning with the HP-UX 10.30 release, the dynamic loader (`dld.sl`) and its application interface library (`libdld.sl`) are thread-safe.

Also, beginning with the HP-UX 10.30 release, the linker toolset provides thread local storage support in:

- `ld` — the link editor
- `dld.sl` — the shared library dynamic loader
- `crt0.o` — the program startup file

Thread local storage (also called thread-specific data) is data specific to a thread. Each thread has its own copy of the data item.

NOTE A program with thread local storage is only supported on systems running HP-UX 10.30 or later versions of the operating system.

NOTE Use of the `__thread` keyword in a shared library prevents that shared library from being dynamically loaded, that is, loaded by an explicit call to `shl_load()`.

For More Information:

- See your HP compiler documentation to learn how to create thread local storage data items with the `_thread` compiler directive.
- See *Programming with Threads on HP-UX* for information on threads.

3 Linker Tasks

You have a great deal of control over how the linker links your program or library by using `ld` command-line options.

- Using the Compiler Command
  - “Changing the Default Library Search Path with -Wl, -L”
  - “Getting Verbose Output with -v”
  - “Passing Linker Options from the Compiler Command with -Wl”
  - “Renaming the Output File with -o”
  - “Specifying Libraries with -l”
  - “Suppressing the Link-Edit Phase with -c”
- Using the Linker Command
  - “Linking with the 32-bit crt0.o Startup File”
  - “Changing the Default Library Search Path with -L and LPATH”
  - “Changing the Default Shared Library Binding with -B”
  - “Choosing Archive or Shared Libraries with -a”
  - “Dynamic Linking with -A and -R”
  - “Exporting Symbols with +e”
  - “Exporting Symbols from main with -E”
  - “Getting Verbose Output with -v”
  - “Hiding Symbols with -h”
  - “Improving Shared Library Performance with -B symbolic”
  - “Moving Libraries after Linking with +b”
  - “Moving Libraries After Linking with +s and SHLIB_PATH”
  - “Passing Linker Options from the Compiler Command with -Wl”
  - “Passing Linker Options in a file with -c”
  - “Passing Linker Options with LDOPTS”

  - “Specifying Libraries with -l and l:”
  - “Stripping Symbol Table Information from the Output File with -s and -x”
- Using the 64-bit mode linker command
  - “Using the 64-bit Mode Linker with +compat or +std”
  - “Linking Shared Libraries with -dynamic”
  - “Linking Archived Libraries with -noshared”
  - “Controlling Archive Library Loading with +[no]forceload”
  - “Flagging Unsatisfied Symbols with +[no]allowunsats”
  - “Hiding Symbols from export with +hideallsymbols”
  - “Changing Mapfiles with -k and +nodefaultmap”
  - “Changing Mapfiles with -k and +nodefaultmap”
  - “Ignoring Dynamic Path Environment Variables with +noenvvar”
  - “Linking in 64-bit Mode with +std”
  - “Linking in 32-bit Mode Style with +compat”
  - “Controlling Output from the Unwind Table with +stripwind”
  - “Selecting Verbose Output with +vtype”
  - “Linking with the 64-bit crt0.o Startup File”
- Linker Compatibility Warnings

Using the Compiler to Link

In many cases, you use your compiler command to compile and link programs. Your compiler uses options that directly affect the linker.

Changing the Default Library Search Path with -Wl, -L

By default, the linker searches the directory `/usr/lib` and `/usr/ccs/lib` for libraries specified with the `-l` compiler option. (If the `-p` or `-G` compiler option is specified, then the linker also searches the profiling library directory `/usr/lib/libp`.)

The `-L` *libpath* option to `ld` augments the default search path; that is, it causes `ld` to search the specified *libpath* before the default places. The C compiler (`cc`), the C++ compiler (`CC`), the POSIX FORTRAN compiler (`fort77`), and the HP Fortran 90 compiler (`f90`) recognize the `-L` option and pass it directly to `ld`. However, the HP FORTRAN compiler (`f77`) and Pascal compiler (`pc`) do not recognize `-L`; it must be passed to `ld` with the `-Wl` option.

Example Using -Wl, -L

For example, to make the `f77` compiler search `/usr/local/lib` to find a locally developed library named `liblocal`, use this command line:

```
$f77 prog.f -Wl,-L,/usr/local/lib -llocal
```

(The `f77` compiler searches `/opt/fortran/lib` and `/usr/lib` as default directories.)

To make the `f90` compiler search `/usr/local/lib` to find a locally developed library named `liblocal`,, use this command line:

```
$f90 prog.f90 -L/usr/local/lib -llocal
```

(The `f90` compiler searches /`opt/fortran90/lib` and `/usr/lib` as default directories.)

For the C compiler, use this command line:

```
$ cc -Aa prog.c -L /usr/local/lib -llocal
```

The LPATH environment variable provides another way to override the default search path. For details, see "Changing the Default Library Search Path with -L and LPATH".

Getting Verbose Output with -v

The `-v` option makes a compiler display verbose information. This is useful for seeing how the compiler calls `ld`. For example, using the `-v` option with the Pascal compiler shows that it automatically links with `libcl`, `libm`, and `libc`.

```
$ pc -v prog.p
/opt/pascal/lbin/pascomp prog.p prog.o -O0
LPATH = /usr/lib/pa1.1:/usr/lib:/opt/langtools/lib
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -z prog.o -lcl -lm -lc
unlink prog.o
```

Passing Linker Options from the Compiler Command with -Wl

The `-Wl` option passes options and arguments to `ld` directly, without the compiler interpreting the options. Its syntax is:

`-Wl,`*arg1* `[`,*arg2*`]`…

where each *argn* is an option or argument passed to the linker. For example, to make `ld` use the archive version of a library instead of the shared, you must specify `-a archive` on the `ld` command line before the library.

Example Using -Wl

The command for telling the linker to use an archive version of `libm` *from the C command line* is:

```
$ cc -Aa mathprog.c -Wl,-a,archive,-lm,-a,default
```

The command for telling the linker to use an archive version of `libm` is:

```
$ ld /opt/langtools/lib/crt0.o mathprog.o -a archive -lm \
  -a default -lc
```

Renaming the Output File with -o

The `-o` *name* option causes `ld` to name the output file *name* instead of `a.out`. For example, to compile a C program `prog.c` and name the resulting file `sum_num`:

```
$ cc -Aa -o sum_num prog.c      Compile using -o option.
$ sum_num                       Run the program.
Enter a number to sum: 5
The sum of 1 to 5: 15
```

Specifying Libraries with -l

Sometimes programs call routines not contained in the default libraries. In such cases you must explicitly specify the necessary libraries on the compile line with the `-l` option. The compilers pass `-l` options directly to the linker *before* the default libraries.

For example, if a C program calls library routines in the curses library (`libcurses`), you must specify `-lcurses` on the `cc` command line:

```
$ cc -Aa -v cursesprog.c -lcurses
    ...
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -u main \
 cursesprog.o -lcurses -lc
cc: Entering Link editor.
```

Linking with the crt0.o Startup File in 32-bit mode

Notice also, in the above example, that the compiler linked `cursesprog.o` with the file `/opt/langtools/lib/crt0.o`. This file contains object code that performs tasks which must be executed when a program starts running — for example, retrieving any arguments specified on the command line when the program is invoked. For details on this file, see *crt0*(3) and “The crt0.o Startup File” on page 43.

Suppressing the Link-Edit Phase with -c

The `-c` compiler option suppresses the link-edit phase. That is, the compiler generates only the `.o` files and *not* the `a.out` file. This is useful when compiling source files that contain only subprograms and data. These may be linked later with other object files, or placed in an archive or shared library. The resulting object files can then be specified on the compiler command line, just like source files. For example:

```
$ f77 -c func.f          Produce .o for func.f.
$ ls func.o
func.o                   Verify that func.o was created.
$ f77 main.f func.o      Compile main.f with func.o
$ a.out                  Run it to verify it worked.
```

Using Linker commands

This section describes linker commands for the 32-bit and 64-bit linker.

NOTE Unless otherwise noted, all examples show 32-bit behavior.

Linking with the 32-bit crt0.o Startup File

In 32-bit mode, you must always include `crt0.o` on the link line.

In 64-bit mode, you must include `crt0.o` on the link line for all fully archive links (`ld -noshared`) and in compatibility mode (`+compat`). You do not need to include the `crt0.o` startup file on the `ld` command line for shared bound links. In 64-bit mode, the dynamic loader, `dld.sl`, does some of the startup duties previously done by `crt0.o`.

See “The crt0.o Startup File” on page 43, and the *crt0*(3) man page for more information.

Changing the Default Library Search Path with -L and LPATH

You can change or override the default linker search path by using the LPATH environment variable or the `-L` linker option.

Overriding the Default Linker Search Path with LPATH

The LPATH environment variable allows you to specify which directories `ld` should search. If LPATH is *not* set, `ld` searches the default directory `/usr/lib`. If LPATH *is* set, `ld` searches only the directories specified in LPATH; the default directories are not searched unless they are specified in LPATH.

If set, LPATH should contain a list of colon-separated directory path names `ld` should search. For example, to include `/usr/local/lib` in the search path after the default directories, set LPATH as follows:

```
$ LPATH=/usr/lib:/usr/local/lib      Korn and Bourne shell syntax.
$ export LPATH
```

Augmenting the Default Linker Search Path with -L

The `-L` option to `ld` also allows you to add additional directories to the search path. If `-L` *libpath* is specified, `ld` searches the *libpath* directory *before* the default places.

For example, suppose you have a locally developed version of `libc`, which resides in the directory `/usr/local/lib`. To make `ld` find this version of `libc` before the default `libc`, use the `-L` option as follows:

```
$ ld /opt/langtools/lib/crt0.o prog.o -L /usr/local/lib -lc
```

Multiple `-L` options can be specified. For example, to search `/usr/contrib/lib` and `/usr/local/lib` before the default places:

```
$ ld /opt/langtools/lib/crt0.o prog.o -L /usr/contrib/lib  \
-L /usr/local/lib -lc
```

If `LPATH` is set, then the `-L` option specifies the directories to search before the directories specified in `LPATH`.

Changing the Default Shared Library Binding with -B

You might want to force **immediate binding** — that is, force all routines and data to be bound at startup time. With immediate binding, the overhead of binding occurs only at program startup, rather than across the program's execution. One possibly useful characteristic of immediate binding is that it causes any possible unresolved symbols to be detected at startup time, rather than during program execution. Another use of immediate binding is to get better interactive performance, if you don't mind program startup taking a little longer.

Example Using -B immediate

To force immediate binding, link an application with the `-B immediate` linker option. For example, to force immediate binding of all symbols in the main program and in all shared libraries linked with it, you could use this `ld` command:

```
$ ld -B immediate /opt/langtools/lib/crt0.o prog.o -lc -lm
```

Nonfatal Shared Library Binding with -B nonfatal

The linker also supports **nonfatal binding**, which is useful with the `-B immediate` option. Like immediate binding, nonfatal immediate binding causes all required symbols to be bound at program startup. The main difference from immediate binding is that program execution continues *even if the dynamic loader cannot resolve symbols.* Compare this with immediate binding, where unresolved symbols cause the program to abort.

To use nonfatal binding, specify the `-B nonfatal` option along with the `-B immediate` option on the linker command line. The order of the options is not important, nor is the placement of the options on the line. For example, the following `ld` command uses nonfatal immediate binding:

```
$ ld /opt/langtools/lib/crt0.o prog.o -B nonfatal \
  -B immediate -lm -lc
```

Note that the `-B nonfatal` modifier does *not* work with deferred binding because a symbol must have been bound by the time a program actually references or calls it. A program attempting to call or access a nonexistent symbol is a fatal error.

Restricted Shared Library Binding with -B restricted

The linker also supports **restricted binding**, which is useful with the `-B deferred` and `-B nonfatal` options. The `-B restricted` option causes the dynamic loader to restrict the search for symbols to those that were visible when the library was loaded. If the dynamic loader cannot find a symbol within the restricted set, a run-time symbol binding error occurs and the program aborts.

The `-B nonfatal` modifier alters this behavior slightly: If the dynamic loader cannot find a symbol in the restricted set, it looks in the global symbol set (the symbols defined in *all* libraries) to resolve the symbol. If it still cannot find the symbol, then a run-time symbol-binding error occurs and the program aborts.

When is `-B restricted` most useful? Consider a program that creates duplicate symbol definitions by either of these methods:

- The program uses `shl_load` with the `BIND_FIRST` flag to load a library that contains symbol definitions that are already defined in a library that was loaded at program startup.

- The program calls `shl_definesym` to define a symbol that is already defined in a library that was loaded at program startup.

If such a program is linked with `-B immediate`, references to symbols will be bound at program startup, regardless of whether duplicate symbols are created later by `shl_load` or `shl_definesym`.

But what happens when, to take advantage of the performance benefits of deferred binding, the same program is linked with `-B deferred`? If a duplicate, more visible symbol definition is created *prior* to referencing the symbol, it binds to the more visible definition, and the program might run incorrectly. In such cases, `-B restricted` is useful, because symbols bind the same way as they do with `-B immediate`, but actual binding is still deferred.

Improving Shared Library Performance with -B symbolic

The linker supports the `-B symbolic` option which optimizes call paths between procedures when building shared libraries. It does this by building direct internal call paths inside a shared library. In linker terms, **import** and **export stubs** are bypassed for calls within the library.

A benefit of `-B symbolic` is that it can help improve application performance and the resulting shared library will be slightly smaller. The `-B symbolic` option is useful for applications that make a lot of calls between procedures inside a shared library *and* when these same procedures are called by programs outside of the shared library.

NOTE

The `-B symbolic` option applies only to function, but not variable, references in a shared library.

Example Using -B symbolic

For example, to optimize the call path between procedures when building a shared library called `lib1.sl`, use `-B symbolic` as follows:

```
$ ld -B symbolic -b func1.o func2.o -o lib1.sl
```

NOTE

The +e option overrides the `-B symbolic` option. For example, you use +e *symbol*, only *symbol* is exported and *all* other symbols are hidden. Similarly, if you use +ee *symbol*, only *symbol* is exported, but other symbols exported by default remain visible.

Since all internal calls inside the shared library are resolved inside the shared library, user-supplied modules with the *same name* are not seen by routines inside the library. For example, you could not replace internal `libc.sl malloc()` calls with your own version of `malloc()` if `libc.sl` was linked with `-B symbolic`.

Comparing -B symbolic with -h and +e

Similar to the -h (hide symbol) and +e (export symbol) linker options, -B symbolic optimizes call paths in a shared library. However, unlike -h and +e, all functions in a shared library linked with -B symbolic are also visible outside of the shared library.

Case 1: Building a Shared Library with -B symbolic.

Suppose you have two functions to place in a shared library. The convert_rtn() calls gal_to_liter().

1. Build the shared library with -b. Optimize the call path inside the shared library with -B symbolic.

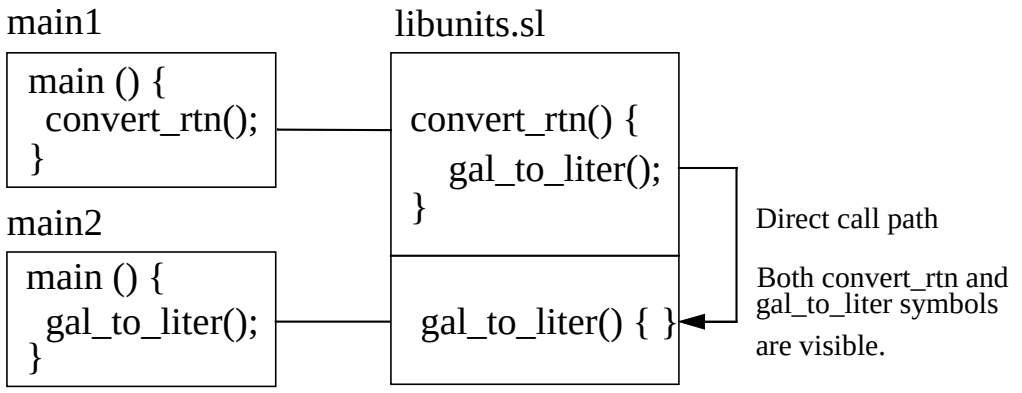
   ```
   $ ld -B symbolic -b convert.o volume.o -o libunits.sl
   ```

2. Two main programs link to the shared library. main1 calls convert_rtn() and main2 calls gal_to_liter().

   ```
   $ cc -Aa main1.c libunits.sl -o main1
   $ cc -Aa main1.c libunits.sl -o main2
   ```

Figure 3-1 shows that a direct call path is established between convert_rtn() and gal_to_liter() inside the shared library. Both symbols are visible to outside callers.

Figure 3-1 Symbols inside a Shared Library Visible with -B symbolic



Case 2: Building a Shared Library with -h or +e. The `-h` (hide symbol) and `+e` (export symbol) options can also optimize the call path in a shared library for symbols that are explicitly hidden. However, only the exported symbols are visible outside of the shared library.

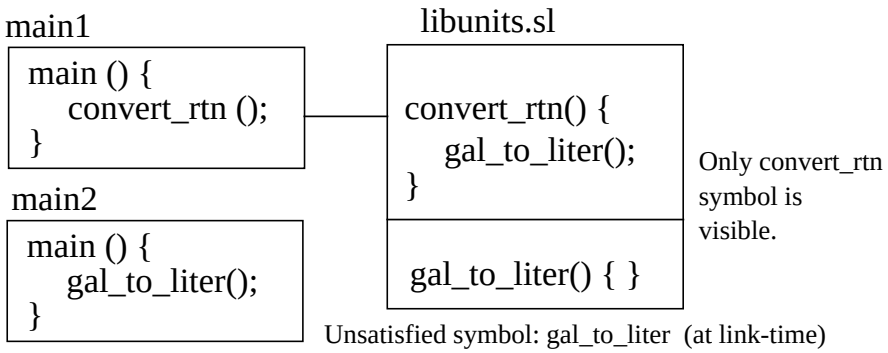
For example, you could hide the `gal_to_liter` symbol as shown:

```
$ ld -b convert.o -h gal_to_liter volume.o -o libunits.sl
```

or export the convert_rtn symbol:

```
$ ld -b +e convert_rtn convert.o volume.o -o libunits.sl
```

In both cases, main2 will not be able to resolve its reference to gal_to_liter() because only the `convert_rtn()` symbol is exported as shown below:



Choosing Archive or Shared Libraries with -a

If both an archive and shared version of a particular library reside in the same directory, `ld` links with the shared version. Occasionally, you might want to override this behavior.

As an example, suppose you write an application that will run on a system on which shared libraries may not be present. Since the program could not run without the shared library, it would be best to link with the archive library, resulting in executable code that contains the required library routines. See also “Caution When Mixing Shared and Archive Libraries” on page 164.

Option Settings to -a

The `-a` option tells the linker what kind of library to link with. It applies to all libraries (`-l` options) until the end of the command line or until the next `-a` option. Its syntax is:

```
-a {archive | shared | default | archive_shared | shared_archive}
```

The different option settings are:

`-a archive`	Select archive libraries. If the archive library does not exist, `ld` generates an error message and does not generate the output file.
`-a shared`	Select shared libraries. If the shared library does not exist, `ld` generates an error message and does not generate the output file.
`-a default`	This is the same as `-a shared_archive`.
`-a archive_shared`	Select the archive library if it exists; otherwise, select the shared library. If the library cannot be found in either version, `ld` generates an error message and does not generate the output file.
`-a shared_archive`	Select the shared library if it exists; otherwise, select the archive library. If the library cannot be found in either version, `ld` generates an error message and does not generate the output file.

The `-a shared` and `-a archive` options specify only one type of library to use. An error results if that type is not found. The other three options specify a preferred type of library and an alternate type of library if the preferred type is not found.

CAUTION

You should avoid mixing shared libraries and archive libraries in the same application. For more information see “Caution When Mixing Shared and Archive Libraries” on page 164.

Example Using -a

The following command links with the archive versions of `libcurses`, `libm` and `libc`:

```
$ ld /opt/langtools/lib/crt0.o prog.o -a archive -lcurses -lm -lc
```

Dynamic Linking with -A and -R

This section describes how to do **dynamic linking** — that is, how to add an object module to a running program. Conceptually, it is very similar to loading a shared library and accessing its symbols (routines and data). In fact, if you require such functionality, you should probably use shared library management routines (see Chapter 6, “Shared Library Management Routines,” on page 195).

However, be aware that dynamic linking is incompatible with shared libraries. That is, a running program *cannot* be linked to shared libraries and also use `ld -A` to dynamically load object modules.

NOTE

Another reason to use shared library management routines instead of dynamic linking is that dynamic linking may not be supported in a future release. See “Linker Compatibility Warnings” and “Changes in Future Releases” on page 32for additional future changes.

Topics in this section include:

- “Overview of Dynamic Linking” describes steps to load an object file into a running program.
- “An Example Program” provides an example dynamic linking scenario.

Overview of Dynamic Linking

The implementation details of dynamic linking vary across platforms. To load an object module into the address space of a running program, and to be able to access its procedures and data, follow these steps on all HP9000 computers:

1. Determine how much space is required to load the module.
2. Allocate the required memory and obtain its starting address.
3. Link the module from the running application.
4. Get information about the module's text, data, and bss segments from the module's header.
5. Read the text and data into the allocated space.
6. Clear (fill with zeros) the bss segment.
7. Flush the text from the data cache before executing code from the loaded module.

8. Get the addresses of routines and data that are referenced in the module.

Step 1: Determine how much space is required to load the module. There must be enough contiguous memory to hold the module's text, data, and bss segments. You can make a liberal guess as to how much memory is needed, and hope that you've guessed correctly. Or you can be more precise by pre-linking the module and getting size information from its header.

Step 2: Allocate the required memory and obtain its starting address. Typically, you use *malloc*(3C) to allocate the required memory. You must modify the starting address returned by `malloc` to ensure that it starts on a memory page boundary (address MOD 4096 == 0).

Step 3: Link the module from the running application. Use the following options when invoking the linker from the program:

`-o` *mod_name*	Name of the output module that will be loaded by the running program.
`-A` *base_prog*	Tells the linker to prepare the output file for incremental loading. Also causes the linker to include symbol table information from *base_prog* in the output file.
`-R` *hex_addr*	Specifies the hexadecimal address at which the module will be loaded. This is the address calculated in Step 2.
`-N`	Causes the data segment to be placed immediately after the text segment.
`-e` *entry_pt*	If specified (it is *optional*), causes the symbol named *entry_pt* to be the entry point into the module. The location of the entry point is stored in the module's header.

Step 4: Get information about the module's text, data, and bss segments from the module's header. There are two header structures stored at the start of the file: `struct header` (defined in `<filehdr.h>`) and `struct som_exec_auxhdr` (defined in `<aouthdr.h>`). The required information is stored in the second header, so to get it, a program must seek past the first header before reading the second one.

The useful members of the `som_exec_auxhdr` structure are:

`.exec_tsize`	Size of text (code) segment.
`.exec_tmem`	Address at which to load the text (already adjusted for offset specified by the `-R` linker option).
`.exec_tfile`	Offset into file (location) where text segment starts.
`.exec_dsize`	Size of data segment.
`.exec_dmem`	Address at which to load the data (already adjusted).
`.exec_dfile`	Offset into file (location) where data segment starts.
`.exec_bsize`	Size of bss segment. It is assumed to start immediately after the data segment.
`.exec_entry`	Address of entry point (if one was specified by the `-e` linker option).

Step 5: Read the text and data into the allocated space. Once you know the location of the required segments in the file, you can read them into the area allocated in Step 2.

The location of the text and data segments in the file is defined by the `.exec_tfile` and `.exec_dfile` members of the `som_exec_auxhdr` structure. The address at which to place the segments in the allocated memory is defined by the `.exec_tmem` and `.exec_dmem` members. The size of the segments to read in is defined by the `.exec_tsize` and `.exec_dsize` members.

Step 6: Clear (zero out) the bss segment. The bss segment starts immediately after the data segment. To zero out the bss, find the end of the data segment and use `memset` (see *memory*(3C)) to zero out the size of the bss.

The end of the data segment can be determined by adding the `.exec_dmem` and `.exec_dsize` members of the `som_exec_auxhdr` structure. The bss's size is defined by the `.exec_bsize` member.

Step 7: Flush the text from the data cache before executing code from the loaded module. Before executing code in the allocated space, a program should flush the instruction and data caches. Although this is really only necessary on systems that have instruction and data caches, it is easiest just to do it on all systems for ease of portability.

Use an assembly language routine named `flush_cache` (see "The flush_cache Function" in this chapter). You must assemble this routine separately (with the `as` command) and link it with the main program.

Step 8: Get the addresses of routines and data that are referenced in the module. If the `-e` linker option was used, the module's header will contain the address of the entry point. The entry point's address is stored in the `.exec_entry` member of the `som_exec_auxhdr` structure.

If the module contains multiple routines and data that must be accessed from the main program, the main program can use the *nlist*(3C) function to get their addresses.

Another approach that can be used is to have the entry point routine return the addresses of required routines and data.

An Example Program

To illustrate dynamic linking concepts, this section presents an example program, `dynprog`. This program loads an object module named `dynobj.o`, which is created by dynamically linking two object files `file1.o` and `file2.o` (see "file1.o and file2.o").

The program allocates space for `dynobj.o` by calling a function named `alloc_load_space` (see "The alloc_load_space Function" later in this chapter). The program then calls a function named `dyn_load` to dynamically link and load `dynobj.o` (see "The dyn_load Function" later in this chapter). Both functions are defined in a file called `dynload.c` (see "dynload.c").

As a return value, `dyn_load` provides the address of the entry point in `dynobj.o` — in this case, the function `foo`. To get the addresses of the function `bar` and the variable `counter`, the program uses the *nlist*(3C) function.

- "The Build Environment" shows the example makefile used to create the `dynprog` program.
- "Source for dynprog" shows the C source code for the `dynprog` program.
- "Output of dynprog" shows the run time output of the `dynprog` program.
- "The flush_cache Function" provides example source code in assembly language to flush text from the data cache.

The Build Environment. Before seeing the program's source code, it may help to see how the program and the various object files were built. The following shows the makefile used to generate the various files.

Makefile Used to Create Dynamic Link Files

```
CFLAGS = -Aa -D_POSIX_SOURCE
dynprog:        dynprog.o dynload.o flush_cache.o
# Compile line:
  cc -o dynprog dynprog.o dynload.o flush_cache.o -Wl,-a,archive

file1.o:        file1.c dynprog.c
file2.o:        file2.c

# Create flush_cache.o:
flush_cache.o:
        as flush_cache.s
```

This makefile assumes that the following files are found in the current directory:

`dynload.c` The file containing the `alloc_load_space` and `dyn_load` functions.

`dynprog.c` The main program that calls functions from `dynload.c` and dynamically links and loads `file1.o` and `file2.o`. Also contains the function `glorp`, which is called by `foo` and `bar`.

`file1.c` Contains the functions `foo` and `bar`.

`file2.c` Contains the variable `counter`, which is incremented by `foo`, `bar`, and `main`.

`flush_cache.s` Assembly language source for function `flush_cache`, which is called by the `dyn_load` function.

To create the executable program `dynprog` from this makefile, you would simply run:

```
$ make dynprog file1.o file2.o
        cc -Aa -D_POSIX_SOURCE -c dynprog.c
        cc -Aa -D_POSIX_SOURCE -c dynload.c
        cc -o dynprog dynprog.o dynload.o -Wl,-a,archive
        cc -Aa -D_POSIX_SOURCE -c file1.c
        cc -Aa -D_POSIX_SOURCE -c file2.c
        as -o flush_cache flush_cache.s
```

Note that the line CFLAGS =... causes any C files to be compiled in ANSI mode (-Aa) and causes the compiler to search for routines that are defined in the Posix standard (-D_POSIX_SOURCE).

For details on using make refer to *make*(1).

Source for dynprog. Here is the source file for the dynprog program.

dynprog.c — Example Dynamic Link and Load Program

```
#include <stdio.h>
#include <nlist.h>

extern void * alloc_load_space(const char * base_prog,
                               const char * obj_files,
                               const char * dest_file);


extern void * dyn_load(const char * base_prog,
                       unsigned int addr,
                       const char * obj_files,
                       const char * dest_file,
                       const char * entry_pt);

const char * base_prog = “dynprog”;      /* this executable’s name
*/
const char * obj_files = “file1.o file2.o”; /* .o files to combine
*/
const char * dest_file = “dynobj.o”;       /* .o file to load
*/
const char * entry_pt  = “foo”;           /* define entry pt name
*/

void glorp (const char *); /* prototype for local
function          */
void (* foo_ptr) ();      /* pointer to entry point foo
*/
void (* bar_ptr) ();      /* pointer to function bar
*/
int * counter_ptr;       /* pointer to variable counter [file2.c]
*/
main()
{
  unsigned int addr;       /* address at which to load dynobj.o  */
  struct nlist nl[3];      /* nlist struct to retrieve addresse  */

/*
STEP 1:  Allocate space for module:
*/
 addr = (unsigned int) alloc_load_space(base_prog,
                                        obj_files, dest_file);

/*
STEP 2:  Load the file at the address, and get address of entry
point:
*/
```

```
  foo_ptr = (void (*)()) dyn_load(base_prog, addr, obj_files,
                                  dest_file, entry_pt);


/*
STEP 3:  Get the addresses of all desired routines using
nlist(3C):
*/

  nl[0].n_name = “bar”;
  nl[1].n_name = “counter”;
  nl[2].n_name = NULL;
  if (nlist(dest_file, nl)) {
    fprintf(stderr, “error obtaining namelist for %s\n”,
dest_file);
    exit(1);
  }
  /*
   * Assign the addresses to meaningful variable names:
   */
  bar_ptr = (void (*)()) nl[0].n_value;
  counter_ptr = (int *) nl[1].n_value;

  /*
   * Now you can call the routines and modify the variables:
   */
  glorp(“main”);
  (*foo_ptr) ();
  (*bar_ptr) ();
  (*counter_ptr) ++;
  printf(“counter = %d\n”, *counter_ptr);
}

void    glorp(const char * from)
{
  printf(“glorp called from %s\n”, from);
}
```

file1.o and file2.o . “Source for file1.c and file2.c” shows the source for `file1.o` and `file2.o`. Notice that `foo` and `bar` call `glorp` in `dynprog.c`. Also, both functions update the variable `counter` in `file2.o`; however, `foo` updates `counter` through the pointer (`counter_ptr`) defined in `dynprog.c`.

Source for file1.c and file2.c

```
/*************************************************************
 * file1.c - Contains routines foo() and bar().

*************************************************************/

extern int * counter_ptr;            /* defined in dynprog.c */
extern int counter;                  /* defined in file2.c   */
extern void glorp(const char * from); /* defined in dynprog.c */

void foo()
{
```

```
 glorp("foo");
 (*counter_ptr) ++; /* update counter indirectly with global
pointer */
}

void bar()
{
 glorp("bar");
 counter ++;        /* update counter directly */
}

/*************************************************************
 * file2.c - Global counter variable referenced by dynprog.c
 * and file1.c.

*************************************************************/
int counter = 0;
```

Output of dynprog . Now that you see how the main program and the module it loads are organized, here is the output produced when dynprog runs:

```
glorp called from main
glorp called from foo
glorp called from bar
counter = 3
```

dynload.c . The dynload.c file contains the definitions of the functions alloc_load_space and dyn_load. "Include Directives for dynload.c" shows the #include directives must appear at the start of this file.

Include Directives for dynload.c

```
#include <stdio.h>
#include <stdlib.h>
#include <nlist.h>
#  include <filehdr.h>
#  include <aouthdr.h>
#  define PAGE_SIZE 4096         /* memory page size  */
```

The alloc_load_space Function . The alloc_load_space function returns a pointer to space (allocated by malloc) into which dynprog will load the object module dynobj.o. It syntax is:

```
void * alloc_load_space(const char * base_prog,
                        const char * obj_files,
                        const char * dest_file)
```

base_prog The name of the program that is calling the routine. In other words, the name of the program that will dynamically link and load *dest_file*.

obj_files The name of the object file or files that will be linked together to create *dest_file*.

dest_file The name of the resulting object module that will by dynamically linked and loaded by *base_prog*.

As described in Step 1 in "Overview of Dynamic Linking" at the start of this section, you can either guess at how much space will be required to load a module, or you can try to be more accurate. The advantage of the former approach is that it is much easier and probably adequate in most cases; the advantage of the latter is that it results in less memory fragmentation and could be a better approach if you have multiple modules to load throughout the course of program execution.

The `alloc_load_space` function allocates only the required amount of space. To determine how much memory is required, `alloc_load_space` performs these steps:

1. Pre-link the specified *obj_files* to create *base_prog*.
2. Get text, data, and bss segment location and size information to determine how much space to allocate.
3. Return a pointer to the space. (The address of the space is adjusted to begin on a memory page boundary — that is, a 4096-byte boundary.)

"C Source for alloc_load_space Function" shows the source for this function.

C Source for alloc_load_space Function

```
void * alloc_load_space(const char * base_prog,
                        const char * obj_files,
                        const char * dest_file)
{
  char cmd_buf[256];    /* linker command line                    */
  int ret_val;          /* value returned by various lib calls    */
  size_t space;         /* size of space to allocate for module   */
  size_t addr;          /* address of allocated space             */
  size_t bss_size;      /* size of bss (uninitialized data)       */
  FILE * destfp;        /* file pointer for dest_file             */

  struct som_exec_auxhdr aux_hdr;       /* file header            */
  unsigned int tdb_size; /* size of text, data, and bss combined */

/* ------------------------------------------------------------
 * STEP 1:  Pre-link the destination module so we can get its size:
 */
  sprintf(cmd_buf, "/bin/ld -a archive -R80000 -A %s -N %s -o %s -lc",
          base_prog, obj_files, dest_file);
  if (ret_val = system(cmd_buf)) {
```

```
    fprintf(stderr, "link failed: %s\n", cmd_buf);
    exit(ret_val);
  }
/* ---------------------------------------------------------------
 * STEP 2:  Get the size of the module's text, data, and bss segments
 * from the auxiliary header for dest_file; add them together to
 * determine size:
*/
  if ((destfp = fopen(dest_file, "r")) == NULL) {
    fprintf(stderr, "error opening %s\n", dest_file);
    exit(1);
  }

  /*
   * Must seek past SOM "header" to get to the desired
   * "som_exec_auxhdr":
   */
  if (fseek(destfp, sizeof(struct header), 0)) {
    fprintf(stderr, "error seeking past header in %s\n", dest_file);
    exit(1);
  }
  if (fread(&aux_hdr, sizeof(aux_hdr), 1, destfp) <= 0) {
    fprintf(stderr, "error reading som aux header from %s\n", dest_file);
    exit(1);
  }

  /* allow for page-alignment of data segment */

  space = aux_hdr.exec_tsize + aux_hdr.exec_dsize
    + aux_hdr.exec_bsize + 2 * PAGE_SIZE;

  fclose(destfp);              /* done reading from module file */
/* ---------------------------------------------------------------
 * STEP 3:  Call malloc(3C) to allocate the required memory and get
 * its address; then return a pointer to the space:
 */
  addr = (size_t) malloc(space);
  /*
   * Make sure allocated area is on page-aligned address:
   */
  if (addr % PAGE_SIZE != 0) addr += PAGE_SIZE - (addr % PAGE_SIZE);

  return((void *) addr);
}
```

The dyn_load Function . The dyn_load function dynamically links and loads an object module into the space allocated by the alloc_load_space function. In addition, it returns the address of the entry point in the loaded module. Its syntax is:

```
void * dyn_load(const char * base_prog,
                unsigned int addr,
                const char * obj_files,
                const char * dest_file,
                const char * entry_pt)
```

The *base_prog*, *obj_files*, and *dest_file* parameters are the same parameters supplied to `alloc_load_space`. The *addr* parameter is the address returned by `alloc_load_space`, and the *entry_pt* parameter specifies a symbol name that you want to act as the entry point in the module.

To dynamically link and load *dest_file* into *base_prog*, the `dyn_load` function performs these steps:

1. Dynamically link *base_prog* with *obj_files*, producing *dest_file*. The address at which *dest_file* will be loaded into memory is specified with the `-R` *addr* option. The name of the entry point for the file is specified with `-e` *entry_pt*.
2. Open *dest_file* and get its header information on the text, data, and bss segments. Read this information into a `som_exec_auxhdr` structure, which starts immediately after a `header` structure.
3. Read the text and data segments into the area allocated by `alloc_load_space`. (The text and data segments are read separately.)
4. Initialize (fill with zeros) the bss, which starts immediately after the data segment.
5. Flush text from the data cache before execution, using the `flush_cache` routine. (See "The flush_cache Function" later in this chapter.)
6. Return a pointer to the entry point, specified by the `-e` option in Step 1.

C Source for dyn_load Function

```
void * dyn_load(const char * base_prog,
                unsigned int addr,

                const char * obj_files,
                const char * dest_file,
                const char * entry_pt)
{
  char  cmd_buf[256];       /* buffer holding linker command
*/
  int   ret_val;           /* holds return value of library calls
*/
  FILE  * destfp;          /* file pointer for destination file
*/
  unsigned int bss_start;   /* start address of bss in VM
*/
  unsigned int bss_size;    /* size of bss                        */
  unsigned int entry_pt_addr; /* address of entry point
```

```
*/

  struct som_exec_auxhdr aux_hdr;  /* som file auxiliary header
*/
  unsigned int tdb_size;    /* size of text, data, and bss
combined*/

/*
---------------------------------------------------------------
 * STEP 1: Dynamically link the module to be loaded:
 */
  sprintf(cmd_buf,
          "/bin/ld -a archive -A %s -R %x -N %s -o %s -lc -e %s",
          base_prog, addr, obj_files, dest_file, entry_pt);

  if (ret_val = system(cmd_buf))
  {
    fprintf(stderr, "link command failed: %s\n", cmd_buf);
    exit(ret_val);
  }

/*
---------------------------------------------------------------
 * STEP 2: Open dest_file. Read its auxiliary header for text,
data,
 *         and bss info:
 */
  if ((destfp = fopen(dest_file, "r")) == NULL)
  {
    fprintf(stderr, "error opening %s for loading\n", dest_file);
    exit(1);
  }

  /*
   * Get auxiliary header information from "som_exec_auxhdr"
struct,
   * which is after SOM header.
   */

  if (fseek(destfp, sizeof(struct header), 0))
  {
    fprintf(stderr, "error seeking past header in %s\n",
dest_file);
    exit(1);
  }

  if (fread(&aux_hdr, sizeof(aux_hdr), 1, destfp) <= 0)
  {
    fprintf(stderr, "error reading som aux header from %s\n",
dest_file);
    exit(1);
  }
/*
---------------------------------------------------------------
 * STEP 3:  Read the text and data segments into the buffer area:
 */

  /*
   * Read text and data separately.  First load the text:
   */
```

```
  if (fseek(destfp, aux_hdr.exec_tfile, 0))
  {
    fprintf(stderr, “error seeking start of text in %s\n”,
dest_file);
    exit(1);
  }

  if ((fread(aux_hdr.exec_tmem, aux_hdr.exec_tsize, 1, destfp)) <=
0)
  {
    fprintf(stderr, “error reading text from %s\n”, dest_file);
    exit(1);
  }
  /*
   * Now load the data, if any:
   */
  if (aux_hdr.exec_dsize) {
    if (fseek(destfp, aux_hdr.exec_dfile, 0))
    {
      fprintf(stderr, “error seeking start of data in %s\n”,
dest_file);
      exit(1);
    }

    if ((fread(aux_hdr.exec_dmem, aux_hdr.exec_dsize, 1,
destfp))<= 0)
    {
      fprintf(stderr, “error reading data from %s\n”, dest_file);
      exit(1);
    }
  }

  fclose(destfp);              /* done reading from module file */
/*
-----------------------------------------------------------------
 * STEP 4:  Zero out the bss (uninitialized data segment):
 */

  bss_start = aux_hdr.exec_dmem + aux_hdr.exec_dsize;
  bss_size  = aux_hdr.exec_bsize;

  memset(bss_start, 0, bss_size);

/*
-----------------------------------------------------------------
 * STEP 5:  Flush the text from the data cache before execution:
 */

  /*
   * The flush_cache routine must know the exact size of the
   * text, data, and bss, computed as follows:
   *   Size = (Data Addr - Text Addr) + Data Size + BSS Size
   * where (Data Addr - Text Addr) = Text Size + alignment between
   *   Text and Data.
   */
  tdb_size = (aux_hdr.exec_dmem - aux_hdr.exec_tmem) +
    aux_hdr.exec_dsize + aux_hdr.exec_bsize;
  flush_cache(addr, tdb_size);
```

```
/*
---------------------------------------------------------------------
 * STEP 6:  Return a pointer to the entry point specified by -e:
 */

  entry_pt_addr = (unsigned int) aux_hdr.exec_entry;
  return ((void *) entry_pt_addr);
}
```

The flush_cache Function . Since there is no existing routine to flush text from the data cache before execution, you must create one. Below is the assembly language source for such a function.

Assembly Language Source for flush_cache Function

```
; flush_cache.s
;
; Routine to flush and synchronize data and instruction caches
; for dynamic loading
;
; Copyright Hewlett-Packard Co. 1985,1991, 1995
;
; All HP VARs and HP customers have a non-exclusive royalty-free
; license to copy and use this flush_cashe() routine in source
; code and/or object code.

        .code

; flush_cache(addr, len) - executes FDC and FIC instructions for
; every cache line in the text region given by starting addr and
; len. When done, it executes a SYNC instruction and then enough
; NOPs to assure the cache has been flushed.
;
; Assumption: Cache line size is at least 16 bytes.  Seven NOPs
; is enough to assure cache has been flushed.  This routine is
; called to flush the cache for just-loaded dynamically linked
; code which will be executed from SR5 (data) space.

; %arg0=GR26, %arg1=GR25, %arg2=GR24, %arg3=GR23, %sr0=SR0.
; loop1 flushes data cache.  arg0 holds address.  arg1 holds
offset.
; SR=0 means that SID of data area is used for fdc.
; loop2 flushes inst cache.  arg2 holds address.  arg3 holds
offset.
; SR=sr0 means that SID of data area is used for fic.
; fdc x(0,y) -> 0 means use SID of data area.
; fic x(%sr0,y) -> SR0 means use SR0 SID (which is set to data
area).

  .proc
  .callinfo
  .export flush_cache,entry
flush_cache
  .enter
  ldsid   (0,%arg0),%r1          ; Extract SID (SR5) from address
  mtsp    %r1,%sr0               ; SID -> SR0
  ldo     -1(%arg1),%arg1        ; offset = length -1
  copy    %arg0,%arg2            ; Copy address from GR26 to GR24
```

```
  copy    %arg1,%arg3             ; Copy offset from GR25 to GR23

  fdc     %arg1(0,%arg0)          ; Flush data cache
@SID.address+offset
loop1
  addib,>,n     -16,%arg1,loop1   ; Decrement offset by cache line
size
  fdc     %arg1(0,%arg0)          ; Flush data cache
@SID.address+offset
  ; flush first word at addr, to handle arbitrary cache line
boundary
  fdc     0(0,%arg0)
  sync

  fic     %arg3(%sr0,%arg2)       ; Flush inst cache
@SID.address+offset
loop2
  addib,>,n      -16,%arg3,loop2 ; Decrement offset by cache line
size
  fic     %arg3(%sr0,%arg2)       ; Flush inst cache
@SID.address+offset
  ; flush first word at addr, to handle arbitrary cache line
boundary
  fic     0(%sr0,%arg2)

  sync
  nop
  nop
  nop
  nop
  nop
  nop
  nop
  .leave
  .procend
  .end
```

Exporting Symbols with +e

The +e option allow you to hide and export symbols. **Exporting** a symbol makes the symbol a global definition, which can be accessed by any other object modules or libraries. The +e option exports *symbol* and hides from export all other global symbols not specified with +e. In essence, -h and +e provide two different ways to do the same thing.

The syntax of the +e option is:

+e *symbol*

Example Using +e

Suppose you want to build a shared library from an object file that contains the following symbol definitions as displayed by the nm command:

```
$ nm -p sem.o
0000000000 U  $global$
1073741824 d  $THIS_DATA$
1073741864 b  $THIS_BSS$
0000000004 cS sem_val
0000000000 T  check_sem_val
0000000036 T  foo
0000000000 U  printf
0000000088 T  bar
0000000140 T  sem
```

In this example, `check_sem_val`, `foo`, `bar`, and `sem` are all global definitions. To create a shared library where `check_sem_val` is a hidden, local definition, you could use either of the following commands:

```
$ ld -b -h check_sem_val sem.o          One -h option.
$ ld -b +e foo +e bar +e sem sem.o      Three +e options.
```

In contrast, suppose you want to export only the `check_sem_val` symbol. Either of the following commands would work:

```
$ ld -b -h foo -h bar -h sem sem.o      Three -h options.
$ ld -b +e check_sem_val sem.o                  One +e option.
```

When to use -h versus +e

How do you decide whether to use `-h` or `+e`? In general, use `-h` if you simply want to hide a few symbols. And use `+e` if you want to export a few symbols and hide a large number of symbols.

You should not combine `-h` and `+e` options on the same command line. For instance, suppose you specify `+e sem`. This would export the symbol `sem` and hide all other symbols. Any additional `-h` options would be unnecessary. If both `-h` and `+e` are used on the same symbol, the `-h` overrides the `+e` option.

The linker command line could get quite lengthy and difficult to read if several such options were specified. And in fact, you could exceed the maximum HP-UX command line length if you specify too many options. To get around this, use `ld` linker option files, described under “Passing Linker Options in a file with -c”. You can specify any number of `-h` or `+e` options in this file.

You can use `-h` or `+e` options when building a shared library (with `-b`) and when linking to create an `a.out` file. When combining `.o` files with `-r`, you can still use only the `-h` option.

Exporting Symbols with +ee

Like the +e option, the +ee option allows you to export symbols. Unlike the +e option, the option does not alter the visibility of any other symbols in the file. It exports the specified symbol, and does not hide any of the symbols exported by default.

Exporting Symbols from main with -E

By default, the linker exports from a program only those symbols that were imported by a shared library. For example, if a shared executable's libraries do *not* reference the program's `main` routine, the linker does *not* include the `main` symbol in the `a.out` file's export list. Normally, this is a problem only when a program calls shared library management routines (described inChapter 6, “Shared Library Management Routines,” on page 195). To make the linker export *all* symbols from a program, invoke `ld` with the `-E` option.

The +e option allows you to be more selective about which symbols are exported, resulting in better performance. For details on +e, see “Exporting Symbols with +e”.

Hiding Symbols with -h

The `-h` option allows you to hide symbols. **Hiding** a symbol makes the symbol a local definition, accessible only from the object module or library in which it is defined. Use `-h` if you simply want to hide a few symbols.

You can use `-h` option when building a shared library (with `-b`) and when linking to create an `a.out` file. When combining `.o` files with `-r`, you can use the `-h` option.

The syntax of the `-h` option is:

`-h` *symbol*

The `-h` option hides *symbol*. Any other global symbols remain exported unless hidden with `-h`.

Example Using -h

Suppose you want to build a shared library from an object file that contains the following symbol definitions as displayed by the `nm` command:

```
$ nm -p sem.o
0000000000 U  $global$
1073741824 d  $THIS_DATA$
1073741864 b  $THIS_BSS$
0000000004 cS sem_val
0000000000 T  check_sem_val
0000000036 T  foo
0000000000 U  printf
0000000088 T  bar
0000000140 T  sem
```

In this example, `check_sem_val`, `foo`, `bar`, and `sem` are all global definitions. To create a shared library where `check_sem_val` is a hidden, local definition, you could do the following:

```
$ ld -b -h check_sem_val sem.o
```

Tips on Using -h

You should not combine `-h` and `+e` options on the same command line. For instance, suppose you specify `+e sem`. This would export the symbol `sem` and hide all other symbols. Any additional `-h` options would be unnecessary. If both `-h` and `+e` are used on the same symbol, the `-h` overrides the `+e` option.

The linker command line could get quite lengthy and difficult to read if several such options were specified. And in fact, you could exceed the maximum HP-UX command line length if you specify too many options. To get around this, use `ld` linker option files, described under "Passing Linker Options in a file with -c". You can specify any number of `-h` or `+e` options in this file.

Hiding and Exporting Symbols When Building a Shared Library

When building a shared library, you might want to hide a symbol in the library for several reasons:

- It can *improve performance* because the dynamic loader does not have to bind hidden symbols. Since most symbols need not be exported from a shared library, hiding selected symbols can have a significant impact on performance.
- It ensures that the definition can only be accessed by other routines in the same library. When linking with other object modules or libraries, the definition will be hidden from them.

- When linking with other libraries (to create an executable), it ensures that the library will use the local definition of a routine rather than a definition that occurs earlier in the link order.

Exporting a symbol is necessary if the symbol must be accessible outside the shared library. But remember that, by default, most symbols are global definitions anyway, so it is seldom necessary to explicitly export symbols. In C, all functions and global variables that are not explicitly declared as `static` have global definitions, while `static` functions and variables have local definitions. In FORTRAN, global definitions are generated for all subroutines, functions, and initialized common blocks.

When using `+e`, be sure to export any data symbols defined in the shared library that will be used by another shared library or the program, even if these other files have definitions of the data symbols. Otherwise, your shared library will use its own private copy of the global data, and another library or the program file will not see any change.

One example of a data symbol that should almost always be exported from a shared library is `errno`. `errno` is defined in every shared library and program; if this definition is hidden, the value of `errno` will not be shared outside of the library.

Hiding Symbols When Combining .o Files with the -r Option

The `-r` option combines multiple `.o` files, creating a single `.o` file. The reasons for hiding symbols in a `.o` file are the same as the reasons listed above for shared libraries. However, a performance improvement will occur only if the resulting `.o` file is later linked into a shared library.

Hiding and Exporting Symbols When Creating an a.out File

By default, the linker exports all of a program's global definitions that are imported by shared libraries specified on the linker command line. For example, given the following linker command, all global symbols in `crt0.o` and `prog.o` that are referenced by `libm` or `libc` are automatically exported:

```
$ ld /usr/ccs/lib/crt0.o prog.o -lm -lc
```

With libraries that are explicitly loaded with `shl_load`, this behavior may not always be sufficient because the linker does not search explicitly loaded libraries (they aren't even present on the command line). You can work around this using the `-E` or `+e` linker option.

As mentioned previously in the section “Exporting Symbols from main with -E”, the `-E` option forces the export of *all* symbols from the program, regardless of whether they are referenced by shared libraries on the linker command line. The `+e` option allows you to be more selective in what symbols are exported. You can use `+e` to limit the exported symbols to only those symbols you want to be visible.

For example, the following `ld` command exports the symbols `main` and `foo`. The symbol `main` is referenced by `libc`. The symbol `foo` is referenced at run time by an explicitly loaded library not specified at link time:

```
$ ld /usr/ccs/lib/crt0.o prog.o +e main +e foo -lm -lc -ldld
```

When using `+e`, be sure to export any data symbols defined in the program that may also be defined in explicitly loaded libraries. If a data symbol that a shared library imports is not exported from the program file, the program uses its own copy while the shared library uses a different copy *if* a definition exists outside the program file. In such cases, a shared library might update a global variable needed by the program, but the program would never see the change because it would be referencing its own copy.

One example of a data symbol that should almost always be exported from a program is `errno`. `errno` is defined in every shared library and program; if this definition is hidden, the value of `errno` will not be shared outside of the program in which it is hidden.

Moving Libraries after Linking with +b

A library can be moved even after an application has been linked with it. This is done by providing the executable with a list of directories to search at run time for any required libraries. One way you can store a directory path list in the program is by using the `+b` *path_list* linker option.

Note that dynamic path list search works only for libraries specified with `-l` on the linker command line (for example, `-lfoo`). It won't work for libraries whose full path name is specified (for example, `/usr/contrib/lib/libfoo.sl`). However, it can be enabled for such libraries with the `-l` option to the `chatr` command (see “Changing a Program's Attributes with chatr(1)” on page 104).

Specifying a Path List with +b

The syntax of the +b option is

```
+b path_list
```

where *path_list* is the list of directories you want the dynamic loader to search at run time. For example, the following linker command causes the path `.:/app/lib::` to be stored in the executable. At run time, the dynamic loader would search for `libfoo.sl`, `libm.sl`, and `libc.sl` in the current working directory (`.`), the directory `/app/lib`, and lastly in the location in which the libraries were found at link time (`::`):

```
$ ld /opt/langtools/lib/crt0.o +b .:/app/lib:: prog.o -lfoo \
  -lm -lc
```

If *path_list* is only a single colon, the linker constructs a path list consisting of all the directories specified by `-L`, followed by all the directories specified by the LPATH environment variable. For instance, the following linker command records the path list as `/app/lib:/tmp`:

```
$ LPATH=/tmp ; export LPATH
$ ld /opt/langtools/lib/crt0.o +b : -L/app/lib prog.o -lfoo \
  -lm -lc
```

The Path List

Whether specified as a parameter to +b or set as the value of the SHLIB_PATH environment variable, the path list is simply one or more path names separated by colons (:), just like the syntax of the PATH environment variable. An optional colon can appear at the start and end of the list.

Absolute and relative path names are allowed. Relative paths are searched relative to the program's current working directory at run time.

Remember that a shared library's full path name is stored in the executable. When searching for a library in an absolute or relative path at run time, the dynamic loader uses only the basename of the library path name stored in the executable. For instance, if a program is linked with `/usr/local/lib/libfoo.sl`, and the directory path list contains `/apps/lib:xyz`, the dynamic loader searches for `/apps/lib/libfoo.sl`, then `./xyz/libfoo.sl`.

The full library path name stored in the executable is referred to as the default library path. To cause the dynamic loader to search for the library in the default location, use a null directory path (). When the loader comes to a null directory path, it uses the default shared library path stored in the executable. For instance, if the directory path list in

the previous example were `/apps/lib::xyz`, the dynamic loader would search for `/apps/lib/libfoo.sl`, `/usr/local/lib/libfoo.sl`, then `./xyz/libfoo.sl`.

If the dynamic loader cannot find a required library in any of the directories specified in the path list, it searches for the library in the default location () recorded by the linker.

Moving Libraries After Linking with +s and SHLIB_PATH

A library can be moved even after an application has been linked with it. Linking the program with `+s`, enables the program to use the path list defined by the `SHLIB_PATH` environment variable at run time.

Specifying a Path List with +s and SHLIB_PATH

When a program is linked with `+s`, the dynamic loader will get the library path list from the `SHLIB_PATH` environment variable at run time. This is especially useful for application developers who don't know where the libraries will reside at run time. In such cases, they can have the user or an install script set `SHLIB_PATH` to the correct value.

For More Information:

- "The Path List" provides additional details about the path list to `SHLIB_PATH`.
- "Moving Libraries after Linking with +b" provides another way to move libraries.

Passing Linker Options in a file with -c

The `-c` *file* option causes the linker to read command line options from the specified *file*. This is useful if you have many `-h` or `+e` options to include on the `ld` command line, or if you have to link with numerous object files. For example, suppose you have over a hundred `+e` options that you need when building a shared library. You could place them in a file named `eopts` and force the linker to read options from the file as follows:

```
$ ld -o libmods.sl -b -c eopts mod*.o
$ cat eopts        Display the file.
+e foo
+e bar
```

```
+e reverse_tree
+e preorder_traversal
+e shift_reduce_parse
  .
  .
  .
```

Note that the linker ignores lines in that option file that begin with a pound sign (#). You can use such lines as comment lines or to temporarily disable certain linker options in the file. For instance, the following linker option file for an application contains a disabled `-O` option:

```
# Exporting only the "compress" symbol resulted
# in better run-time performance:
+e compress
# When the program is debugged, remove the pound sign
# from the following optimization option:
# -O
```

Passing Linker Options with LDOPTS

If you use certain linker options all the time, you may find it useful to specify them in the `LDOPTS` environment variable. The linker inserts the value of this variable before all other arguments on the linker command line. For instance, if you always want the linker to display verbose information (`-v`) and a trace of each input file (`-t`), set `LDOPTS` as follows:

```
$ LDOPTS="-v -t"   Korn and Bourne shell syntax.
$ export LDOPTS
```

Thereafter, the following commands would be equivalent:

```
$ ld /opt/langtools/lib/crt0.o -u main prog.o -lc
$ ld -v -t /opt/langtools/lib/crt0.o -u main prog.o -lc
```

Specifying Libraries with -l and l:

To direct the linker to search a particular library, use the `-l`*name* option. For example, to specify `libc`, use `-lc`; to specify `libm`, use `-lm`; to specify `libXm`, use `-lXm`.

Specifying Libraries (-l)

When writing programs that call routines not found in the default libraries linked at compile time, you must specify the libraries on the compiler command line with the `-l`*x* option. For example, if you write a C program that calls POSIX math functions, you must link with `libm`.

The *x* argument corresponds to the identifying portion of the library path name — the part following `lib` and preceding the suffix `.a` or `.sl`. For example, for the `libm.sl` or `libm.a` library, *x* is the letter `m`:

```
$ cc -Aa mathprog.c -lm
```

The linker searches libraries in the order in which they are specified on the command line (that is, the **link order**). In addition, libraries specified with `-l` are searched *before* the libraries that the compiler links by default.

Using the -l: option

The `-l:` option works just like the `-l` option with one major difference: `-l:` allows you to specify the full basename of the library to link with. For instance, `-l:libm.a` causes the linker to link with the archive library `/usr/lib/libm.a`, regardless of whether `-a shared` was specified previously on the linker command line.

The advantage of using this option is that it allows you to specify an archive or shared library explicitly without having to change the state of the `-a` option. (See also “Caution When Mixing Shared and Archive Libraries” on page 164.)

For instance, suppose you use the `LDOPTS` environment variable (see “Passing Linker Options with LDOPTS”) to set the `-a` option that you want to use by default when linking. And depending on what environment you are building an application for, you might set `LDOPTS` to `-a archive` or `-a shared`. You can use `-l:` to ensure that the linker will always link with a particular library regardless of the setting of the `-a` option in the `LDOPTS` variable.

Example Using -l:

For example, even if `LDOPTS` were set to `-a shared`, the following command would link with the archive `libfoo.a` in the directory `/usr/mylibs`, the archive `libm.a` and `libc.a`:

```
$ ld /opt/langtools/lib/crt0.o -u main prog.o -L/usr/mylibs \
     -l:libfoo.a -l:libc.a -l:libm.a
```

Stripping Symbol Table Information from the Output File with -s and -x

The `a.out` file created by the linker contains symbol table, relocation, and (if debug options were specified) information used by the debugger. Such information can be used by other commands that work on `a.out` files, but is not actually necessary to make the file run. `ld` provides two command line options for removing such information and, thus, reducing the size of executables:

`-s` Strips all such information from the file. The executable becomes smaller, but difficult or impossible to use with a symbolic debugger. You can get much the same results by running the `strip` command on an executable (see *strip*(1)). In some cases, however, `-s` rearranges the file to save more space than `strip`.

`-x` Strips *only local symbols* from the symbol table. It reduces executable file size with only a minimal affect on commands that work with executables. However, using this option may still make the file unusable by a symbolic debugger.

These options can reduce the size of executables dramatically. Note, also, that these options can also be used when generating shared libraries without affecting shareability.

Using 64-bit Mode Linker Options

This section introduces 64-bit-only linker options.

Using the 64-bit Mode Linker with +compat or +std

In the HP-UX 11.0 release, the linker toolset supports extended features for linking in 64-bit mode. Since compatibility with the previous linker toolset is a high priority, the 64-bit linker uses much of the old behavior in the new toolset. The 64-bit mode linker includes two options to allow you to instruct the linker to link in one of two modes:

- Compatibility mode, with the `+compat` option, to create a link and operation in 32-bit style. Because of some object file format restrictions, the mode is not completely compatible with the style of the 32-bit linker.
- Standard mode, with the `+std` option, set by default in 64-bit mode, to create a link and load operation in 64-bit style. This mode uses the new behaviors and features of the 64-bit linker.

Using the Linker with +compat for Compatibility Mode

The +compat option instructs the linker to do a 32-bit-style link.

When you use the `+compat` option, the linker:

- Uses 32-bit style shared library internal name processing.
- Lists all dependent shared libraries in a `DT_HP_NEEDED` entry the dynamic table using the 32 bit-style shared library naming conventions. These dependent libraries are recorded as compatibility mode libraries even if they are really created as standard mode dependent libraries.
- If an error occurs during the link, the linker creates an `a.out` without the executable permission bits set.
- Does not use embedded paths at link time to find dependent libraries.
- Considers the order of `ld +b` and `+s`.

- +b first means dld looks at the RPATH first when searching for dependent shared libraries.

  To get the default RPATH, you must specify ld +b. This instructs the linker to construct a default RPATH consisting of the -L directories and LPATH.

- +s first means the dynamic loader looks at the SHLIB_PATH environment variable first when searching for dependent shared libraries.

  You must specify ld +s to force the dynamic loader to use SHLIB_PATH to search for shared libraries at runtime.

At runtime, the dynamic loader does a 32-bit style load for all compatibility mode dependent shared libraries. The dynamic loader:

- Does dynamic path searching for compatibility-mode dependent shared libraries that have the dynamic path selected (set in the DT_HP_NEEDED entry if the shared library was specified with -l).
- Uses SHLIB_PATH only if you specify ld +s (or chatr +s) for compatibility-mode shared libraries.
- Allows RPATH inheritance from ancestors to children when searching for dependent compatibility-mode shared libraries specified with ld -l. This is only allowed in an a.out that was linked with +compat. If the a.out was linked +std, no library (even a compatibility mode shared library) uses embedded RPATH inheritance.
- Allows dynamic path searching on shared libraries loaded by shl_load routines, if the DYNAMIC_FLAG is passed to shl_load().
- Does a depth-first search of all compatibility-mode dependent libraries.
- Looks at RPATH or SHLIB_PATH first, depending on the ld +b/+s ordering for all ld -l dependent shared libraries. The dynamic loader looks at whichever has second precedence next, and then looks for the shared library as specified in the dynamic load entry.
- Looks for the dynamic table entry as if the dynamic path bit is not set.

Using the 64-bit Linker with +std for Standard Mode

The +std option instructs the linker to do a standard mode 64-bit style link. This is currently the default in 64-bit mode.

This default may change in future releases.

When you use +std, the linker:

- Assumes -dynamic was passed to ld. The linker looks for shared libraries first. The output executable is a shared executable.
- All dependent shared libraries are output in the dynamic table in a DT_NEEDED entry. These dependent shared libraries are recorded as standard mode shared libraries.
- ld +b and +s ordering is ignored. ld +s is on by default.
- If an error occurs during the link, the linker does not generate an a.out file.
- Uses de facto standard internal name processing for dependent shared libraries.
- Uses embedded RPATHs at link time to find dependent shared libraries.
- If you do not specify ld +b, the linker uses a default RPATH consisting of the -L directories, LPATH, and the default directories /usr/lib/pa20_64:/usr/ccs/lib/pa20_64.

At runtime, the dynamic loader does a 64-bit-style load for all standard mode dependent shared libraries. The dynamic loader:

- Does dynamic path searching only for standard-mode shared libraries in the DT_NEEDED entry of the dynamic table which do not contain a path. For those standard-mode dynamic libraries that contain paths, dld looks for the library as specified.
- Looks for the shared library as specified in the DT_NEEDED dynamic table entry if it contains a path.
- Looks at LD_LIBRARY_PATH and SHLIB_PATH environment variables at runtime by default when doing dynamic path searching for standard-mode shared libraries.
- Does not allow RPATH inheritance from ancestors to children (only allowed from parent to child).
- Does a breadth-first search for all standard-mode dependent shared libraries.

- Looks at the environment variables first, followed by RPATH, and the default directories by default when doing dynamic path searching for standard-mode dependentshared libraries.

Linking Shared Libraries with -dynamic

Use the `-dynamic` option to instruct the linker to look for shared libraries first and then archive libraries. The linker outputs a dynamically linked executable.

This option is on by default in standard mode.

In the following example, the linker only looks for shared libraries:

```
$ld main.o -dynamic -L. -lbar -lc
```

If you specified an archive library, the linker links it in, but the resulting executable is still a dynamically linked executable. This is true even if the linker finds no shared libraries at link time.

Linking Archived Libraries with -noshared

Use the `-noshared` option if you need to link with all archive libraries. The linker outputs a statically bound executable.

NOTE You cannot link in shared libraries if you specify this option.

In the following example, the linker only looks for `/usr/lib/pa20_64/libfoo.a` and `/usr/lib/pa20_64/libc.a`:

```
ld crt0.o main.o -noshared -L. -lfoo -lc
```

If you specify a shared library with this option, the linker emits an error message.

```
ld: The shared library “libbar.sl” cannot be processed in a static
link.
Fatal error.
```

Controlling Archive Library Loading with +[no]forceload

Use the `+[no]forceload` option to control how the linker loads object files from an archived library. `+forceload` instructs the linker to load all object files from an archive library. `+noforceload` tells the linker to

only load those modules from an archive library that is needed. The mode you select, either by default or explicitly, remains on until you change it.

+noforceload is the default on both 32-bit and 64-bit modes.

In the following example, main() references foo(), which is a module in mylib.a. foo() doesn't reference any other module in mylib.a and libc.a. If mylib.a contains foo.o and bar.o, then only foo.o is linked in.

```
ld crt0.o main.o +vtype libraries mylib.a -lc

...
Selecting liba.a[foo.o] to resolve foo
ld crt0.o main.o +forceload mylib.a -lc +vtype libraries

...
Selecting liba.a[foo.o] to forcibly load
Selecting liba.a[bar.o] to forcibly load
```

Flagging Unsatisfied Symbols with +[no]allowunsats

Use the +allowunsats option to instruct the linker to not flag unsatisfied symbols at link time. This is the default for relocatable (-r) and shared library builds (-b), and is the default behavior in 32-bit mode.

Use the +noallowunsat option to instruct the linker to flag as an error any unsatisfied symbol in the resulting output file. The linker still creates a.out, but the file does not have any execute permission bits set. This is the default for program files (same behavior as in 32-bit mode).

For example, where main() references functions foo() and bar(). bar() resides in libbar.sl. foo() resides in libfoo.sl

```
ld main.o +allowunsats -L. -lbar -lc

ld: (warning) Unsatisfied symbol “foo”.
1 warning.
```

+allowunsats still causes the linker to emit a warning message and output a.out. If you do not specify the option and the linker finds an unsatisfied symbol, the linker emits an error message and an unexecutable a.out only if linking with +compat set.

```
ld main.o -L. -lbar -lc

ld: Unsatisfied symbol “foo”.
1 error.
```

Hiding Symbols from export with +hideallsymbols

Use the +hideallsymbols option to hide all symbols to prevent the linker from exporting them in a shared link.

In the following example, main() exports func() and test(). Using +hideallsymbols, the linker does not export these two routines in the a.out.

```
ld main.o +hideallsymbols -L. -lfoo -lc
elfdump -t a.out
a.out:
...
.symtab
index Type Bind Other SectValueSizeName
1 FUNC LOCL 00xb 0x4000000000001104 0test
...
10FUNCLOCL00xb0x40000000000012000func
```

Changing Mapfiles with -k and +nodefaultmap

The linker automatically maps sections from input object files onto output segments in executable files. These options to the ld command allow you to change the linker's default mapping.

Use the -k filename option to provide a memory map. The linker uses the file specified by filename as the output file memory map.

The +nodefaultmap option used with -k option prevents the linker from concatenating the default memory map to the map provided by *filename*. If you specify +nodefaultmap, the linker does not append the default mapfile to your mapfile. If you do not specify +nodefaultmap with -k, the linker appends the output file to the default mapfile.

NOTE In most cases, the linker produces a correct executable without the use of the mapfile option. The mapfile option is an advanced feature of the linker toolset intended for systems programming use, not application programming use. When using the mapfile option, you can create executable files that do not execute.

For more information on mapfiles and examples using these options, see Appendix A, "Using Mapfiles," on page 295.

Ignoring Dynamic Path Environment Variables with +noenvvar

Use the +noenvvar to instruct the dynamic loader not to look at the environment variables relating to dynamic path searching at runtime. It ignores LD_LIBRARY_PATH and SHLIB_PATH environment variables. This option is on by default in with ld +compat. It is off by default with ld +std.

For example, if libbar.sl has dependent library libfee.sl that is i ./ at link time, but is moved to /tmp by runtime:

```
ld main.o -L. -lbar -lc
export LD_LIBRARY_PATH=/tmp
mv libbar.sl /tmp
a.out
called bar()
called fee()
mv /tmp/libbar.sl ./
ld main.o +noenvvar -L. -lbar -lc
mv libbar.sl /tmp
a.out
dld.sl: Unable to find library “libbar.sl”
```

Linking in 64-bit Mode with +std

Use the +std option to instructs the linker to do a 64-bit mode link. This is the default mode. For more information, see “Using the 64-bit Mode Linker with +compat or +std”.

Linking in 32-bit Mode Style with +compat

Use the +compat option to instruct the linker to do a 32-bit mode style link. For more information, see “Using the 64-bit Mode Linker with +compat or +std”.

Controlling Output from the Unwind Table with +stripwind

Use the +stripunwind option to suppress output of the unwind table.

```
ld -b foo.o -o libfoo.sl +stripunwind
elfdump -U libfoo.sl

libfoo.sl:
```

Selecting Verbose Output with +vtype

Use the +vtype option to get verbose output about specified elements of the link operation. The following values specify the type:

Parameter	Description
files	Dump information about each object file loaded.

```
ld main.o +vtype files -L. -lfile1 -lfile2 -lc
Loading main.o:
Loading ./libfile1.sl:
Loading ./libfile2.sl:
Loading /usr/lib/pa20_64/libc.2:
Loading /usr/lib/pa20_64/libdl.1:
```

libraries	Dump information about libraries searched.

```
ld main.o +vtype libraries -L. -lfile1 -lfile2 -lc
Searching /usr/lib/pa20_64/libc.a:
Selecting /usr/lib/pa20_64/libc.a[printf.o] to
resolve printf
Selecting /usr/lib/pa20_64/libc.a[data.o] to
resolve __iob
...
```

sections	Dump information about each section added to the output file.

```
ld main.o +vtype sections -L. -lfile1 -lfile2 -lc
main.o:
section .text PROG_BITS AX 116 8 added to text
segment
section .PARISC.unwind UNWIND 16 4 added to text
segment
section .data PROG_BITS AW 96 8 added to data
segment
```

symbols	Dump information about global symbols referenced/defined from/in the input files.

```
ld main.o +vtype symbols -L. -lfile1 -lfile2 -lc
main.o:
main is DEFINED GLOBAL FUNC
printf is UNDEF GLOBAL FUNC
lib1_func is UNDEF GLOBAL FUNC
lib2_func is UNDEF GLOBAL FUNC
./libfile1.s:
printf is UNDEF GLOBAL FUNC
_DYNAMIC is DEFINED GLOBAL OBJECT
lib1_func is DEFINED GLOBAL FUNC
...
```

all	Dump all of the above. Same as -v.

```
ld main.o +vtype all -L. -lfile1 -lfile2 -lc
Loading main.o:
main.o:
main is DEFINED GLOBAL FUNC
printf is UNDEF GLOBAL FUNC
lib1_func is UNDEF GLOBAL FUNC
lib2_func is UNDEF GLOBAL FUNC
main.o:

section .text PROG_BITS AX 116 8 added to text
segment
section .PARISC.unwind UNWIND 16 4 added to text
segment
section .data PROG_BITS AW 96 8 added to data
segment
Loading ./libfile1.sl:
./libfile1.sl:
...
```

Linking with the 64-bit crt0.o Startup File

In 32-bit mode, you must always include `crt0.o` on the link line.

In 64-bit mode, you must include `crt0.o` on the link line for all fully archive links (`ld -noshared`) and in compatibility mode (`+compat`). You do not need to include the `crt0.o` startup file on the `ld` command line for shared bound links. In 64-bit mode, the dynamic loader, `dld.sl`, does some of the startup duties previously done by `crt0.o`.

See “The crt0.o Startup File” on page 43, and *crt0*(3) manual page for more information.

Linker Compatibility Warnings

Beginning with the HP-UX 10.20 release, the linker generates compatibility warnings. These warnings include HP 9000 architecture issues, as well as linker features that may change over time. Compatibility warnings can be turned off with the `+vnocompatwarnings` linker option. Also, detailed warnings can be turned on with the `+vallcompatwarnings` linker option. See the *ld*(1) man page for a description of these options.

Link-time compatibility warnings include the following:

- *Linking PA-RISC 2.0 object files on any system* — PA-RISC 1.0 programs run on 1.1 and 2.0 systems. PA-RISC 2.0 programs do not run on 1.1 or 1.0 systems. See Also "PA-RISC Changes in Hardware Compatibility" on page 21.
- *Dynamic linking with -A* — If you do dynamic linking with `-A`, you should migrate to using the shared library management routines described in Chapter 6, "Shared Library Management Routines," on page 195. These routines are also described in the *sh_load*(3X) and *dl**(3X) man page.

  The 64-bit mode linker does not support the -A option.
- *Procedure call parameter and return type checking (which can be specified with -C)* — The 32-bit linker checks the number and types of parameters in procedure calls across object modules. In a future release, you should expect HP compilers to perform cross-module type checking, instead of the linker. This impacts HP Pascal and HP Fortran programs.

  The 64-bit mode linker does not support the -C option.
- *Duplicate names found for code and data symbols* — The 32-bit linker can create a program that has a code and data symbol with the same name. In the HP-UX 11.00 release, the 64-bit mode linker adopts a single name space for all symbols. This means that code and data symbols cannot share the same name. Renaming the conflicting symbols solves this problem.

- *Unsatisfied symbols found when linking to archive libraries* — If you specify the `-v` option with the `+vallcompatwarnings` option and link to archive libraries, you may see new warnings. For an example, see "Linking to Archive Libraries with Unsatisfied Symbols" in this chapter.
- *Versioning within a shared library* — If you do versioning within a shared library with the `HP_SHLIB_VERSION` (C and C++); or the `SHLIB_VERSION` (Fortran and Pascal) compiler directive, you should migrate to the industry standard and faster-performing library-level versioning. See "Library-Level Versioning" on page 150 to learn how to do library-level versioning. In the HP-UX 11.00 release, the 64-bit mode linker does not support internal library versioning.

Linking to Archive Libraries with Unsatisfied Symbols

If you link a program that contains a reference to an archive library, and the archive library contains an undefined symbol, you may see the following warning:

```
ld: (Warning) The file library.a(x.o) has not been fully
checked for unsatisfied symbols.  This behavior may
change in future releases.
```

The 32-bit mode linker does not include an object from an archive library simply because it contains a needed definition of an uninitialized global data symbol. Instead, it changes the existing undefined symbol to an uninitialized data symbol. This symbol has the same size as the definition of the global variable in the library.

For example, given these source files:

`archive.c`

```
int foo;        /* definition of uninitialized
                   global data symbol              */
void func()
{
     unsat();
}
```

`main.c`

```
extern int foo; /* declaration of global data symbol */
main()
{
```

```
     printf ("\tfoo = %d\n", foo);
}
```

If these files are compiled and linked as:

```
cc -c main.c
cc -c archive.c
ar rv liba.a archive.o
ld /opt/langtools/lib/crt0.o -v \
  +vallcompatwarnings main.o liba.a -lc -o test
```

The linker issues the following warning:

```
ld: (Warning) The file liba.a(archive.o) has not been fully
checked for unsatisfied symbols. This behavior may change
in future releases.
```

due to an unresolved symbol for `unsat()`.

In the HP-UX 11.00 release, the linker includes the archive library object definition rather than fixing up the external reference.

4 Linker Tools

This chapter describes the linker toolset, which provides several tools to help you find symbols, display and modify object files, and determine link order. Some of these tools are specific to a particular object file type; others are available in both 32-bit and 64-bit mode.

The following table lists the linker toolset.

Tool	Mode	Description
`chatr`	32-bit/ 64-bit	Displays or modifies the internal attributes of an object file. See “Changing a Program's Attributes with chatr(1)”.
`elfdump`	64-bit	Displays the contents of an ELF object file. See “Viewing the Contents of an Object File with elfdump(1)”.
`fastbind`	32-bit/ 64-bit	Improves startup time of programs that use shared libraries. See “Improving Program Start-up with fastbind(1)”.
`ldd`	64-bit	Lists dynamic dependencies of executable files and shared libraries. “Viewing library dependencies with ldd(1)”.
`lorder`	32-bit/ 64-bit	Finds ordering relationship for an object library. See“Finding Object Library Ordering Relationships with lorder(1)”.
`nm`	32-bit/ 64-bit	Displays the symbol table of an object file. See “Viewing Symbols in an Object file with nm(1)”.
`size`	32-bit/ 64-bit	Prints sizes of object file elements. See “Viewing the Size of Object File Elements with size(1)”.
`strip`	32-bit/ 64-bit	Strips symbol and debugging information from an object file, executable, or archive library. See “Reducing Storage Space with strip(1)”.

Changing a Program's Attributes with chatr(1)

The `chatr` command (see *chatr*(1)) allows you to change various program attributes that were determined at link time. When run without any options, `chatr` displays the attributes of the specified file.

Using chatr for 32-bit Program Attributes

The following table summarizes the options you can use to change various attributes:

To do this:	Use this option:
32-bit mode only: Set the file's magic number to `SHARE_MAGIC`.	`-n`
32-bit mode only: Set the file's magic number to `DEMAND_MAGIC`.	`-q`
32-bit mode only: Change the file's magic number from `EXEC_MAGIC` to `SHMEM_MAGIC`.	`-M`
32-bit mode only: Change the file's magic number from `SHMEM_MAGIC` to `EXEC_MAGIC`.	`-N`
Use immediate binding for all libraries loaded at program startup.	`-B immediate`
Use deferred binding for all libraries loaded at program startup.	`-B deferred`
Use nonfatal binding. Must be specified with `-B immediate` or `-B deferred`.	`-B nonfatal`
Use restricted binding. Must be specified with `-B immediate` or `-B deferred`.	`-B restricted`
Enable run-time use of the path list specified with the `+b` option at link time.	`+b enable` [a]
Disable run-time use of the path list specified with the `+b` option at link time.	`+b disable`

To do this:	Use this option:
Enable the use of the SHLIB_PATH environment variable to perform run-time path list lookup of shared libraries.	`+s enable`[a]
Disable the use of the SHLIB_PATH environment variable to perform run-time path list lookup of shared libraries.	`+s disable`
32-bit mode only: Do *not* subject a library to path list lookup, even if path lists are provided. That is, use default library path stored in the executable.	`+l` *libname*
32-bit mode only: Subject a library to path list lookup if directory path lists are provided. Useful for libraries that were specified with a full path name at link time.	`-l` *libname*
Set the virtual memory page size for data segments.	`+pd` *size*
Set the virtual memory page size for instructions.	`+pi` *size*
Assist branch prediction on PA-RISC 2.0 systems. Programs must be linked with `+Ostaticprediction`.	`+k`
Request static branch prediction.	`+r`

a. If `+b enable` and `+s enable` are both specified, the order in which they appear determines which search path is used first.

Using chatr for 64-bit Program Attributes

In 64-bit mode, `chatr` supports two different command syntaxes. One is compatible with the 32-bit command. Use it to modify files that have only a single text segment and data segment. The second command syntax allows you specify selected segments to modify. The following sections list the additional 64-bit mode options for the `chatr` command.

For the 32-bit compatible syntax:

To do this:	Use this option:
Set the modification bit for the file's data segment(s).	+md
Set the modification bit for the file's text segment(s).	+mi
Set the code bit for the file's data segment(s).	+cd
Set the code bit for the file's text segment(s).	+ci
Enable lazy swap on all data segments. Do not use with non-data segments.	+z

For the 64-bit only syntax:

To do this:	Use this option:
Set the code bit for a specified segment.	+c
Enables or disables lazy swap allocation for dynamically allocated segments (such as the stack or heap).	+dz
Set the modification bit for a specified segment.	+m
Set the page size for a specified segment.	+p
Identify a segment using a segment index number.	+si
Identify a segment using an address.	+sa
Use all segments in the file for a set of attribute modifications.	+sall
Enable lazy swap on a specific segment (using the second command syntax). Do not use with non-data segments.	+z

Viewing Symbols in an Object file with nm(1)

The *nm*(1) command displays the symbol table of each specified object. *file* can be a relocatable object file or an executable object file, or an archive of relocatable or executable object files.

`nm` provides three general output formats: the default (neither `–p` nor `–P` specified), `–p`, and `–P`. See the *nm*(1) man page for a detailed description of the output formats.

To	Use This Option
Prefix each output line with the name of the object file or archive, file. Equivalent to `–r`.	`–A`
64-bit mode ELF files only: Demangle C++ names before printing them.	`–C`
Display the value and size of a symbol in decimal. This is the default for the default format or the –p format. Equivalent to `-t d`.	`–d`
Display only *external* and *static* symbols. This option is ignored (see `–f`).	`–e`
Display full output. This option is in force by default.	`–f`
Display only *external* (global) symbol information.	`–g`
Do not display the output header data.	`–h`
Distinguish between weak and global symbols by appending * to the key letter of weak symbols. Only takes effect with `-p` and/or `-P`.	`–l`
Sort symbols by name, in ascending collation order, before they are printed. This is the default. To turn off this option, use `-N`.	`–n`
Display symbols in the order in which they appear in the symbol table.	`–N`
Display the value and size of a symbol in octal. Equivalent to `-t o`.	`–o`

To	Use This Option
Display information in a blank-separated output format. Each symbol name is preceded by its value (blanks if undefined) and one of the letters **A** absolute **B** bss symbol **C** common symbol **D** data symbol **R** section region **S** tstorage symbol *(32-bit mode SOM files only)* If the symbol is local (nonexternal), the type letter is in lowercase. If the symbol is a secondary definition, the type letter is followed by the letter S. Note that –p is not compatible with –P. **T** text symbol **U** undefined	–p
Display information in a portable output format as specified below, to standard output. Note that -P is not compatible with -P.	-P
32-bit mode SOM files only: Silence some warning messages.	-q
Prefix each output line with the name of the object file or archive, file. Equivalent to –A.	-r
64-bit mode ELF files only: Print the section index instead of the section name.	-s
Display each numeric value in the specified format. *format* can be one of: **d** Display the value and size of a symbol in decimal. This is the default for the default format or the –p format. Equivalent to -d. **o** Display the value and size of a symbol in octal. Equivalent to -o. **x** Display the value and size of a symbol in hexadecimal. This is the default for the –P format. Equivalent to -x.	–t *format*

To	Use This Option
32-bit mode SOM files only: Truncate every name that would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated. If `-A` or `-r` is also specified, the file prefix is truncated first. By default, `nm` prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names overflows its column, forcing every column after the name to be misaligned.	`-T`
Display undefined symbols only.	`-u`
Print the usage menu.	`-U`
Sort symbols by value before they are printed.	`-v`
Display the executing version of the `nm` command on standard error.	`-V`
Displays the value and size of a symbol in hexadecimal. this is the default for the `-P` format. Equivalent to `-t x`.	`-x`

Examples

- Display which object files have undefined references for the symbol "`leap`":

  ```
  "nm –rup *.o | grep leap"
  ```

- Display which object files have a definition for the text symbol "`leap`":

  ```
  nm –rp *.o | awk '{ if\
  ($3 == " "T" " && $4 == " "leap" ") { print $0 } }'"
  ```

- To view the symbols defined in an object file, use the `nm` command. The following 32-bit mode example shows output from running `nm -p` on the `func.o` and `main.o` object files.

  ```
  $ nm -p func.o
                                        Other symbols created from compiling.
  1073741824 d  $THIS_DATA$
  1073741824 d  $THIS_SHORTDATA$
  1073741824 b  $THIS_BSS$
  1073741824 d  $THIS_SHORTBSS$
  0000000000 T  sum_n                   Global definitions of sum_n.
  $ nm -p main.o
  ```

```
0000000000 U  $global$                Other symbols created from
compiling.
1073741824 d  $THIS_DATA$
1073741872 d  $THIS_SHORTDATA$
1073741872 b  $THIS_BSS$
1073741872 d  $THIS_SHORTBSS$
0000000000 T  main                    Global definition of main.
0000000000 U  printf
0000000000 U  scanf
0000000000 U  sum_n
```

The first column shows the address of each symbol or reference. The last column shows the symbol name. The second column denotes the symbol's type:

`T`	indicates a global definition.
`U`	indicates an external reference.
`d`	indicates a local definition of data.
`b`	indicates a local definition of uninitialized data (bss).

Thus, a global definition of `sum_n` is found in `func.o`. An external reference to `sum_n` is found in `main.o`. External references to the C `printf` and `scanf` routines are found in `main.o`. For details on the use of `nm`, see *nm*(1).

Viewing the Contents of an Object File with elfdump(1)

NOTE The `elfdump` command works on 64-bit executables or shared libraries.

The *elfdump*(1) command displays information contained in ELF format object files, archives, and shared libraries.

Use the following options to select the information you want to display:

To view the contents.	Use this option
Symbol table entries.	`-t`
Archive headers from an archive library.	`-a`
String table(s).	`-c`
File header.	`-f`
Global symbols from an archive.	`-g`
Section headers.	`-h`
The .dynamic section in shared libraries and dynamically linked program files.	`-L`
Optional headers (program headers).	`-o`
Relocations.	`-r`
Section contents.	`-s`
Unwind table.	`-U`

`elfdump` provides the following additional options to modify your selections:

Option	Modifies	Causes elfdump to
`-H`	all	Select output format in hexadecimal, octal, or decimal.
`-p`	all	Suppress title printing.
`-S`	`-h`,`-o`	Display headers in short format.
`-C`	`-c`, `-r`, `-s`, `-t`	Demangle C++ symbol names before displaying them. • With `-H`, ignored. • With `-n` *name*, display the symbol whose unmangled name matches *name*, and prints its symbol name as a demangled name.
`-D` *num*	`-h`, `-s`	Display the section whose index is *num*.
`+D` *num2*	`-h`, `-s`	Display the sections in the range 1 to *num2*. • With `-D`, display the sections in the range *num* to *num2*.
`-D` *num*	`-r`	Display the relocation whose index is *num*.
`+D` *num2*	`-r`	Display only the relocations which apply to the section(s) in the range.
`+s` *name*	`-c`, `-t`	Display the section specified by *name*.
`-n` *name*	`-h`, `-r`, `-s`	Display information about the section specified by *name*.
`-n` *name*	`-t`	Display information about the symbol entry specified by *name*.
`-T` *num*	`-t`	Display the symbol whose index is *num*.
`+T` *num2*	`-t`	Display the symbols in the range 0 to *num2*. • With `-T`, display the symbols in the range *num* to *num2*.

Viewing library dependencies with ldd(1)

NOTE The `ldd` command works on 64-bit executables or shared libraries.

The *ldd*(1) command lists the dynamic dependencies of executable files or shared libraries. `ldd` displays verbose information about dynamic dependencies and symbol references:

Executable All shared libraries that would be loaded as a result of executing the file.

Shared library All shared libraries that would be loaded as a result of loading the library.

`ldd` uses the same algorithm as the dynamic loader (`/usr/lib/pa20_64/dld.sl`) to locate the shared libraries.

`ldd` does not list shared libraries explicitly loaded using `dlopen`(3X) or `shl_load`(3X).

`ldd` prints the record of shared library path names to `stdout`. It prints the optional list of symbol resolution problems to `stderr`.

To do this	Use the option
Check reference to data symbols.	`-d`
Check reference to data and code symbols.	`-r`
Displays the search path used to locate the shared libraries.	`-s`
Display all dependency relationships.	`-v`

Examples

- By default, `ldd` prints simple dynamic path information, including the dependencies recorded in the executable (or the shared library) followed by the physical location where the dynamic loader finds these libraries.

```
$ldd a.out

./libx.sl =>./libx.sl
libc.2 =>/lib/pa20_64/libc.2
libdl.1 =>/lib/pa20_64/libdl.1
```

- The `-v` option causes `ldd` to print dependency relationship along with the dynamic path information.

```
$ldd -v a.out

find library=./libx.sl; required by a.out
./libx.sl =>./libx.sl
find library=libc.2; required by a.out
libc.2 =>/lib/pa20_64/libc.2
find library=libdl.1; required by /lib/pa20_64/libc.2
libdl.1 =>/lib/pa20_64/libdl.1
```

- The `-r` option to causes it to analyze all symbol references and print information about unsatisfied code and data symbols.

```
$ldd -r a.out

./libx.sl=>./libx.sl
libc.2=>/lib/pa20_64/libc.2
libdl.1=>/lib/pa20_64/libdl.1
symbol not found: val1 (./libx.sl)
symbol not found: count (./libx.sl)
symbol not found: func1 (./libx.sl)
symbol not found: func2 (./libx.sl)
```

Viewing the Size of Object File Elements with size(1)

The *size*(1) command produces section size information for each section in your specified object files. It displays the size of the text, data and bss (uninitialized data) sections with the total size of the object file. If you specify an archive file, the information for all archive members is displayed.

Use the following options to display information for your specified files:

To display	Use this option
Sizes in decimal (default).	`-d`
Sizes in octal.	`-o`
Sizes in hexadecimal.	`-x`
Version information about the `size` command.	`-V`
Verbose list of the subspaces in the object files. Each subspace is listed on a separate line with its size, physical address, and virtual address.	`-v`
64-bit mode only: Size of each allocatable section (=.	`-f`
64-bit mode only: Size and permission bits of each loadable segment=.	`-F`
64-bit mode only: Sizes of non loadable segments or non allocatable sections.	`-n`

Reducing Storage Space with strip(1)

The *strip*(1) command removes the symbol table and line number information from object files, including archives. Thereafter, no symbolic debugging access is available for that file. The purpose of this command is to reduce file storage overhead consumed by the object file. Use this command on production modules that have been debugged and tested. The effect is nearly identical to using the `-s` option of `ld`.

You can control the amount of information stripped from the symbol table by using the following options:

To	Use this option
Strip line number information only; do not strip any symbol table information.	`-l`
Do not strip static or external symbol information.	`-x`
32-bit mode only: Reset the relocation indexes into the symbol table. This option allows `strip` to be run on relocatable files, in which case the effect is also to strip only symbolic debugging information and unloadable data.	`-r`
Print the version of the `strip` command to `stderr`.	`-V`

NOTE

The `-l` and `-x` options are synonymous because the symbol table contains only static and external symbols. Either option strips only symbolic debugging information and unloadable data.

If there are any relocation entries in the object file and any symbol table information is to be stripped, `strip` issues a message and terminates without stripping the specified file unless the `-r` option is used.

If you execute `strip` on an archive file (see *ar*(4)), it removes the archive symbol table. The archive symbol table must be restored by executing `ar` with its `s` operator (see *ar*(1)) before the `ld` command (see *ld* (1)) can use the archive. `strip` issues appropriate warning messages when this situation occurs.

Improving Program Start-up with fastbind(1)

The *fastbind*(1) command prepare an incomplete executable for faster program start-up. It can improve the start-up time of programs that use shared libraries (incomplete executables) by storing information about needed shared library symbols in the executable file.

`fastbind` performs analysis on the symbols used to bind an executable and all of its dependent shared libraries, and stores this information in the executable file. The next time the executable is run, the dynamic loader (`/usr/lib/dld.sl` for 32-bit mode or `/usr/lib/pa20_64/dld.sl` for 64-bit mode) detects that this information is available, and uses it to bind the executable instead of using the standard search method for binding the symbols.

Because `fastbind` writes the fastbind information in the executable file, you must have write permission on the executable file. If the executable file being analyzed is being run as another process or the file is locked against modifications by the kernel, the `fastbind` command fails.

If the shared libraries that an executable is dependent on are modified after the fastbind information is created, the dynamic loader silently reverts to standard search method for binding the symbols. The fastbind information can be re–created by running `fastbind` on the executable again. `fastbind` automatically erases the old fastbind information and generate the new one.

To do this	Use this option
Remove the fastbind information from the executable, returning it to the same state it as was in before you ran `fastbind` on it.	`-n`
Normally, if `fastbind` detects any unsatisfied symbols while building the fastbind information, it generates an error message and does not modify the executable file. When you invoke `fastbind` with the `-u` option however, it allows unresolved symbols.	`-u`

The 32-bit mode `fastbind` command does not work with `EXEC_MAGIC` executables.

`fastbind` effectively enforces the binding modes bind-restricted and bind-immediate. For example, consider an executable linked bind-deferred, which calls a function `foo()` defined in an implicitly loaded library. Before the actual call is made, if it explicitly loads a shared library (using `shl_load`(3X) with `BIND_FIRST`) having a definition for `foo()` when `foo()` is finally called, it is resolved from the explicitly-loaded library. But after running `fastbind`, the symbol `foo()` is resolved from the implicitly-loaded library.

For more information about fastbind and performance, see “Improving Shared Library Start-Up Time with fastbind” on page 293.

Example

- To run `fastbind` on the executable file `a.out`:

  $**fastbind a.out**

- To later remove the fastbind information from the executable file `a.out`

  $**fastbind -n a.out**

Finding Object Library Ordering Relationships with lorder(1)

The lorder command finds the ordering relation for an object library. You can specify one or more object or archive library files (see *ar*(1)) on the command line or read those files from standard input. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second.

You can process the output with `tsort` to find an ordering of a library suitable for one-pass access by `ld` (see *tsort*(1) and *ld*(1)). The linker `ld` is capable of multiple passes over an archive in the archive format and does not require that you use `lorder` when building an archive. Using the `lorder` command may, however, allow for a slightly more efficient access of the archive during the link-edit process.

The symbol table maintained by `ar` allows `ld` to randomly access symbols and files in the archive, making the use of `lorder` unnecessary when building archive libraries (see *ar*(1)).

`lorder` overlooks object files whose names do not end with `.o`, even when contained in library archives, and attributes their global symbols and references to some other file.

Examples

- Build a new library from existing `.o` files:

  ```
  $ar cr library 'lorder *.o | tsort'
  ```

- When creating libraries with so many objects that the shell cannot properly handle the `*.o` expansion, the following technique may prove useful:

  ```
  $ls |grep '.o$'|lorder|tsort|xargs ar cq library
  ```

5 Creating and Using Libraries

Many libraries come with HP-UX. You can also create and use your own libraries on HP-UX. This chapter provides information on the following topics:

- General Information about Shared and Archive Libraries
  - “Overview of Shared and Archive Libraries”
  - “What are Archive Libraries?”
  - “What are Shared Libraries?”
  - “Example Program Comparing Shared and Archive Libraries”
  - “Shared Libraries with Debuggers, Profilers, and Static Analysis”
- Creating Libraries on HP-UX
  - “Creating Shared Libraries”
  - “Creating Archive Libraries”
- Using Libraries on HP-UX
  - “Switching from Archive to Shared Libraries”
  - “Summary of HP-UX Libraries”
  - “Caution When Mixing Shared and Archive Libraries”
- Using Shared Libraries in 64-bit Mode
  - “Internal Name Processing”
  - “Dynamic Path Searching for Shared Libraries”
  - “Shared Library Symbol Binding Semantics”
  - “Mixed Mode Shared Libraries”
  - “64-bit Mode Library Examples”

Overview of Shared and Archive Libraries

HP-UX supports two kinds of libraries: **archive** and **shared**. A shared library is also called **a dll** (dynamically linked library), particularly in the context of the 64-bit mode linker. Archive libraries are the more traditional of the two. The following table summarizes differences between archive and shared libraries.

Comparing	Archive	Shared (or dll)
file name suffix	Suffix is `.a`.	Suffix is `.sl` or *.number* representing a particular version of the library.
object code	Made from relocatable object code.	Made from **position-independent** object code, created by compiling with the `+z` or `+Z` compiler option. Can also be created in assembly language (see Chapter 7, “Position-Independent Code,” on page 259).
creation	Combine object files with the `ar` command	Combine PIC object files with the `ld` command

Comparing	Archive	Shared (or dll)
address binding	Addresses of library subroutines and data are resolved at link time.	Addresses of library subroutines are bound at run time. Addresses of data in `a.out` are bound at link time; addresses of data in shared libraries are bound at run time.
`a.out` files	Contains all library routines or data (external references) referenced in the program. An `a.out` file that does not use shared libraries is known as a **complete executable**.	Does not contain library routines; instead, contains a **linkage table** that is filled in with the addresses of routines and shared library data. An `a.out` that uses shared libraries is known as an **incomplete executable**, and is almost always *much* smaller than a complete executable.
run time	Each program has its own copy of archive library routines.	Shared library routines are shared among all processes that use the library.

Almost all system libraries are available both as a shared library and as an archive library for 32-bit mode in the directory `/usr/lib` and for 64-bit mode in `/usr/lib/pa20_lib`. Archive library file names end with `.a` whereas shared library file names end with `.sl`. For example, in 32-bit mode, the archive C library `libc` is `/usr/lib/libc.a` and the shared version is `/usr/lib/libc.sl`. In 64-bit mode, the archive C library `libc` is `/usr/lib/pa20_64/libc.a` and the shared version is `/usr/lib/pa20_64/libc.sl`

If both shared and archived versions of a library exist, `ld` uses the one that it finds first in the default library search path. If both versions exist in the same directory, `ld` uses the shared version. For example, compiling the C program `prog.c` causes `cc` to invoke the linker with a command like this:

- For 32-bit mode: `ld /opt/langtools/lib/crt0.o prog.o -lc`
- For 64-bit mode:

  ```
  ld /opt/langtools/lib/pa20_64/crt0.o prog.o -lc
  ```

The `-lc` option instructs the linker to search the C library, `libc` or `libc/pa20_64`, to resolve unsatisfied references from `prog.o`. If a shared `libc` exists (`/usr/lib/libc.sl` or `/usr/lib/pa20_64/libc.sl`), `ld` uses it instead of the archive `libc`

(`/usr/lib/libc.a` or `/usr/lib/pa20_64/libc.a`). You can, however, override this behavior and select the archive version of a library with the `-a` option or the `-l:` option. These are described in “Choosing Archive or Shared Libraries with -a” on page 63 and “Specifying Libraries with -l and l:” on page 87.

In addition to the system libraries provided on HP-UX, you can create your own archive and shared libraries. To create archive libraries, combine object files with the `ar` command, as described in “Overview of Creating an Archive Library”. To create shared libraries, use `ld` to combine object files containing position-independent code (PIC), as described in “Creating Shared Libraries”.

For more information, see “Caution When Mixing Shared and Archive Libraries”.

What are Archive Libraries?

An **archive library** contains one or more object files and is created with the `ar` command. When linking an object file with an archive library, `ld` searches the library for global definitions that match up with external references in the object file. If a match is found, `ld` copies the object file containing the global definition from the library into the `a.out` file. In short, any routines or data a program needs from the library are copied into the resulting `a.out` file.

NOTE

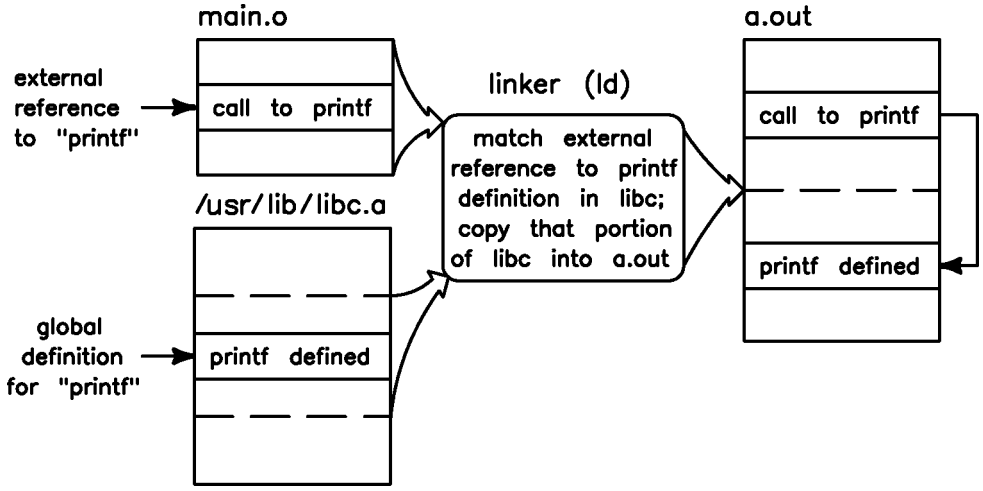
For 32-bit only:

If the only definition referenced in an object file of an archive library is a common symbol, only that common symbol is copied into the `a.out` and not the entire object file. This helps reduce the size of the `a.out` file.

Example

For example, in 32-bit mode, suppose you write a C program that calls `printf` from the `libc` library. "Linking with an Archive Library" shows how the resulting `a.out` file would look if you linked the program with the archive version of `libc`.

Figure 5-1 Linking with an Archive Library



What are Shared Libraries?

Like an archive library, a **shared library** contains object code. However, `ld` treats shared libraries quite differently from archive libraries. When linking an object file with a shared library, `ld` does not copy object code from the library into the `a.out` file; instead, the linker simply notes in the `a.out` file that the code calls a routine in the shared library. An `a.out` file that calls routines in a shared library is known as an incomplete executable.

The Dynamic Loader dld.sl

When an incomplete executable begins execution, the HP-UX **dynamic loader** (see *dld.sl*(5)) looks at the `a.out` file to see what libraries the `a.out` file needs during execution. In 32-bit mode, the startup code in `crt0.o` activates the dynamic loader. In 64-bit mode, the kernel activates the dynamic loader for a 64-bit `a.out`.The dynamic loader then loads and maps any required shared libraries into the process's address space — known as **attaching** the libraries. A program calls shared library routines indirectly through a **linkage table** that the dynamic loader fills in with the addresses of the routines. By default, the dynamic loader places the addresses of shared library routines in the linkage table as the routines are called — known as **deferred binding**. **Immediate binding** is also possible — that is, binding all required symbols in the shared library at program startup. In either case, any routines that are already loaded are shared.

Consequently, linking with shared libraries generally results in smaller `a.out` files than linking with archive libraries. Therefore, a clear benefit of using shared libraries is that it can reduce disk space and virtual memory requirements.

NOTE

In prior releases, data defined by a shared library was copied into the program file at link time. All references to this data, both in the libraries and in the program file, referred to the copy in the executable file.

With the HP-UX 10.0 release, however, this data copying is eliminated. Data is accessed in the shared library itself. The code in the executable file references the shared library data indirectly through a linkage pointer, in the same way that a shared library would reference the data.

Default Behavior When Searching for Libraries at Run Time

By default, if the dynamic loader cannot find a shared library from the list, it generates a run-time error and the program aborts. For example, in 32-bit mode, suppose that during development, a program is linked with the shared library `liblocal.sl` in your current working directory (say, `/users/hyperturbo`):

```
$ ld /opt/langtools/lib/crt0.o prog.o -lc liblocal.sl
```

In 32-bit mode, the linker records the path name of `liblocal.sl` in the `a.out` file as `/users/hyperturbo/liblocal.sl`. When shipping this application to users, you must ensure that (1) they have a copy of `liblocal.sl` on their system, and (2) it is in the same location as it was when you linked the final application. Otherwise, when the users of your application run it, the dynamic loader will look for `/users/hyperturbo/liblocal.sl`, fail to find it, and the program will abort.

In 64-bit mode, the linker records `./liblocal.sl`.

This is more of a concern with non-standard libraries—that is, libraries not found in `/usr/lib` or `/usr/lib/pa20_64`. There is little chance of the standard libraries not being in these directories.

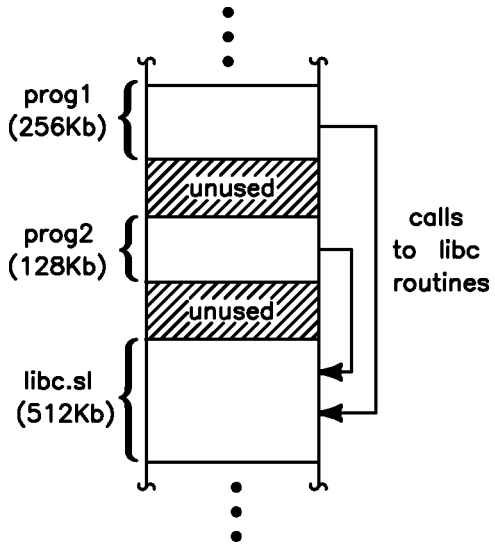
Caution on Using Dynamic Library Searching

If different versions of a library exist on your system, be aware that the dynamic loader may get the wrong version of the library when dynamic library searching is enabled with `SHLIB_PATH` or `+b`. For instance, you may want a program to use the PA1.1 libraries found in the `/usr/lib/pa1.1` directory; but through a combination of `SHLIB_PATH` settings and `+b` options, the dynamic loader ends up loading versions found in `/usr/lib` instead. If this happens, make sure that `SHLIB_PATH` and `+b` are set to avoid such conflicts.

Example Program Comparing Shared and Archive Libraries

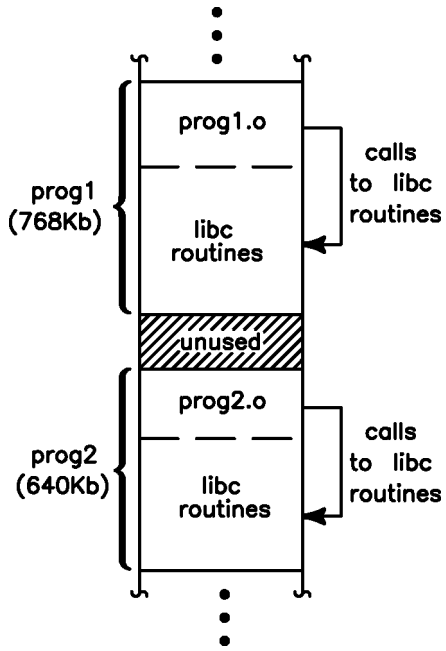
As an example, suppose two separate programs, `prog1` and `prog2`, use shared `libc` routines heavily. Suppose that the `a.out` portion of `prog1` is 256Kb in size, while the `prog2` `a.out` portion is 128Kb. Assume also that the shared `libc` is 512Kb in size. Figure 5-2 shows how physical memory might look when both processes run simultaneously. Notice that one copy of `libc` is shared by both processes. The total memory requirement for these two processes running simultaneously is 896Kb (256Kb + 128Kb + 512Kb).

Figure 5-2 Two Processes Sharing libc



Compare this with the memory requirements if `prog1` and `prog2` had been linked with the archive version of `libc`. As shown in Figure 5-3, 1428Kb of memory are required (768Kb + 640Kb). The numbers in this example are made up, but it is true in general that shared libraries reduce memory requirements.

Figure 5-3 Two Processes with Their Own Copies of libc



Shared Libraries with Debuggers, Profilers, and Static Analysis

As of the HP-UX 10.0 release, debugging of shared libraries is supported by the HP/DDE debugger. For details on how to debug shared libraries, refer to the *HP/DDE Debugger User's Guide*.

Profiling with `prof` and `gprof` and static analysis are not allowed on shared libraries.

Creating Archive Libraries

Two steps are required to create an archive library:

1. Compile one or more source files to create object files containing **relocatable object code**.
2. Combine these object files into a single **archive library** file with the `ar` command.

Shown below are the commands you would use to create an archive library called `libunits.a`:

```
cc -Aa -c length.c volume.c mass.c
ar r libunits.a length.o volume.o mass.o
```

These steps are described in detail in "Overview of Creating an Archive Library".

Other topics relevant to archive libraries are:

- "Contents of an Archive File"
- "Example of Creating an Archive Library"
- "Replacing, Adding, and Deleting an Object Module"
- "Summary of Keys to the ar(1) Command"
- "Archive Library Location"

Overview of Creating an Archive Library

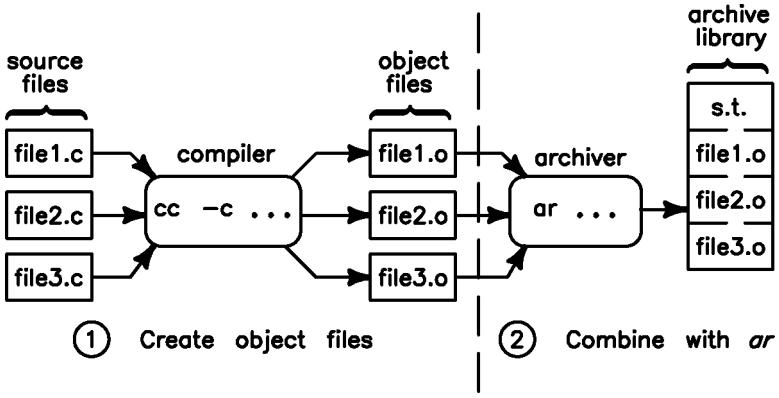
To create an archive library:

1. Create one or more object files containing relocatable object code. Typically, each object file contains one function, procedure, or data structure, but an object file could have multiple routines and data.
2. Combine these object files into a single archive library file with the `ar` command. Invoke `ar` with the `r` key.

   ("Keys" are like command line options, except that they do not require a preceding `-`.)

Figure 5-4 summarizes the procedure for creating archive libraries from three C source files (`file1.c`, `file2.c`, and `file3.c`). The process is identical for other languages, except that you would use a different compiler.

Figure 5-4 Creating an Archive Library



Contents of an Archive File

An archive library file consists of four main pieces:

1. a **header string**, "!<arch>\n", identifying the file as an archive file created by `ar` (`\n` represents the newline character)
2. a **symbol table**, used by the linker and other commands to find the location, size, and other information for each routine or data item contained in the library
3. an optional **string table** used by the linker to store file names that are greater than 15 bytes long (only created if a long file name is encountered)
4. **object modules**, one for each object file specified on the `ar` command line

To see what object modules a library contains, run `ar` with the `t` key, which displays a table of contents. For example, to view the "table of contents" for `libm.a`:

```
$ ar t /usr/lib/libm.a
```
Run ar with the t key.

```
cosh.o
```
Object modules are displayed.

```
erf.o
fabs.o
floor.o
  . . . .
```

This indicates that the library was built from object files named `cosh.o`, `erf.o`, `fabs.o`, `floor.o`, and so forth. In other words, module names are the same as the names of the object files from which they were created.

Example of Creating an Archive Library

Suppose you are working on a program that does several conversions between English and Metric units. The routines that do the conversions are contained in three C-language files shown:

length.c - Routine to Convert Length Units

```
float   in_to_cm(float in)  /* convert inches to centimeters */
{
  return (in * 2.54);
}
```

volume.c - Routine to Convert Volume Units

```
float   gal_to_l(float gal)  /* convert gallons to liters */
{
  return (gal * 3.79);
}
```

mass.c - Routine to Convert Mass Units

```
float   oz_to_g(float oz)    /* convert ounces to grams */
{
       return (oz * 28.35);
}
```

During development, each routine is stored in a separate file. To make the routines easily accessible to other programmers, they should be stored in an archive library. To do this, first compile the source files, either separately or together on the same command line:

```
$ cc -Aa -c length.c volume.c mass.c      Compile them together.
length.c:
volume.c:
mass.c:
$ ls *.o                                  List the .o files.
length.o     mass.o      volume.o
```

Then combine the `.o` files by running `ar` with the `r` key, followed by the library name (say `libunits.a`), followed by the names of the object files to place in the library:

```
$ ar r libunits.a length.o volume.o mass.o
ar: creating libunits.a
```

To verify that `ar` created the library correctly, view its contents:

```
$ ar t libunits.a          Use ar with the t key.
length.o
volume.o
mass.o                     All the .o modules are included; it worked.
```

Now suppose you've written a program, called `convert.c`, that calls several of the routines in the `libunits.a` library. You could compile the main program and link it to `libunits.a` with the following `cc` command:

```
$ cc -Aa convert.c libunits.a
```

Note that the whole library name was given, and the `-l` option was not specified. This is because the library was in the current directory. If you move `libunits.a` to `/usr/lib` before compiling, the following command line will work instead:

```
$ cc -Aa convert.c -lunits
```

Linking with archive libraries is covered in detail in Chapter 3, “Linker Tasks,” on page 51.

Replacing, Adding, and Deleting an Object Module

Occasionally you may want to replace an object module in a library, add an object module to a library, or delete a module completely. For instance, suppose you add some new conversion routines to `length.c` (defined in the previous section) and want to include the new routines in the library `libunits.a`. You would then have to *replace* the `length.o` module in `libunits.a`.

Replacing or Adding an Object Module

To replace or add an object module, use the `r` key (the same key you use to create a library). For example, to replace the `length.o` object module in `libunits.a`:

```
$ ar r libunits.a length.o
```

Deleting an Object Module

To delete an object module from a library, use the `d` key. For example, to delete `volume.o` from `libunits.a`:

```
$ ar d libunits.a volume.o          Delete volume.o.
$ ar t libunits.a                   List the contents.
length.o
mass.o                              volume.o is gone.
```

Summary of Keys to the ar(1) Command

When used to create and manage archive libraries, `ar`'s syntax is:

`ar [-] keys archive [modules] ...`

archive is the name of the archive library. *modules* is an optional list of object modules or files. See *ar*(1) for the complete list of keys and options.

Useful ar Keys

Here are some useful `ar` *keys* and their modifiers:

key	Description
`d`	Delete the *modules* from the *archive*.
`r`	Replace or add the *modules* to the *archive*. If *archive* exists, `ar` replaces *modules* specified on the command line. If *archive* does not exist, `ar` creates a new *archive* containing the *modules*.
`t`	Display a table of contents for the *archive*.
`u`	Used with the `r`, this modifier tells `ar` to replace only those *modules* with creation dates later than those in the *archive*.
`v`	Display verbose output.
`x`	Extracts object modules from the library. Extracted modules are placed in `.o` files in the current directory. Once an object module is extracted, you can use `nm` to view the symbols in the module.

For example, when used with the `v` flag, the `t` flag creates a verbose table of contents — including such information as module creation date and file size:

```
$ ar tv libunits.a
rw-rr   265/    20    230 Feb  2 17:19 1990 length.o
rw-rr   265/    20    228 Feb  2 16:25 1990 mass.o
rw-rr   265/    20    230 Feb  2 16:24 1990 volume.o
```

The next example replaces `length.o` in `libunits.a`, *only if* `length.o` is more recent than the one already contained in `libunits.a`:

```
$ ar ru libunits.a length.o
```

crt0.o

In 64-bit mode, the `crt0.o` startup file is not needed for shared bound links because `dld.sl` does some of the startup duties previously done by `crt0.o`. You still need to include `crt0.o` on the link line for all fully archive links (`ld -noshared`). In 32-bit mode, `crt0.o` must always be included on the link line.

Users who link by letting the compilers such as `cc` invoke the linker do not have include `crt0.o` on the link line.

Archive Library Location

After creating an archive library, you will probably want to save it in a location that is easily accessible to other programmers who might want to use it. There are two main choices for places to put the library:

- in the 32-bit `/usr/lib` or 64-bit `/user/lib/pa20_64` directory
- in the 32-bit `/usr/local/lib` or `/usr/contrib/lib` directory

Using /usr/lib and /usr/lib/pa20_64

Since the linker searches `/usr/lib` or `/usr/lib/pa20_64` by default, you might want to put your archive libraries there. You would eliminate the task of entering the entire library path name each time you compile or link.

The drawbacks of putting the libraries in `/usr/lib` or `/usr/lib/pa20_64` are:

- You usually need super-user (system administrator) privileges to write to the directory.
- You could overwrite an HP-UX system library that resides in the directory.

Check with your system administrator before attempting to use `/usr/lib` or `/usr/lib/pa20_64`.

Using /usr/local/lib or /usr/contrib/lib

The `/usr/local/lib` and `/usr/local/lib/pa20_64` library typically contain libraries created locally — by programmers on the system; `/usr/contrib/lib` and `/usr/contrib/lib/pa20_64` contain libraries supplied with HP-UX but not supported by Hewlett-Packard. Programmers can create their own libraries for both 32-bit and 64-bit code using the same library name. Although `ld` does not automatically search these directories, they are still often the best choice for locating user-defined libraries because the directories are not write-protected. Therefore, programmers can store the libraries in these directories without super-user privileges. Use `-L/usr/local/lib` or `-L/usr/contrib/lib` for 32-bit libraries, or `-L/usr/local/lib/pa20_64` or `-L/usr/contrib/lib/pa20_64` for 64-bit libraries to tell the linker to search these directories.

Creating Shared Libraries

Two steps are required to create a shared library:

1. "Creating Position-Independent Code (PIC)" by compiling with `+z`.
2. "Creating the Shared Library with ld" by linking with `-b`.

Shown below are the commands you would use to create a shared library called `libunits.sl`:

```
$ cc -Aa -c +z length.c volume.c mass.c
$ ld -b -o libunits.sl length.o volume.o mass.o
```

Other topics relevant to shared libraries are:

- "Shared Library Dependencies"
- "Updating a Shared Library"
- "Version Control with Shared Libraries"
- "Shared Library Location"
- "Improving Shared Library Performance"

Creating Position-Independent Code (PIC)

The first step in creating a shared library is to create object files containing **position-independent code** (**PIC**). There are two ways to create PIC object files:

- Compile source files with the `+z` or `+Z` compiler option, described below.
- Write assembly language programs that use appropriate addressing modes, described in Chapter 7, "Position-Independent Code," on page 259.

In 32-bit mode, the `+z` and `+Z` options force the compiler to generate PIC object files. In 64-bit mode, the `+Z` option is the default.

Example Using +z

Suppose you have some C functions, stored in `length.c`, that convert between English and Metric length units. To compile these routines and create PIC object files with the C compiler, you could use this command:

```
$ cc -Aa -c +z length.c        The +z option creates PIC.
```

You could then link it with other PIC object files to create a shared library, as discussed in "Creating the Shared Library with ld".

Comparing +z and +Z

In 32-bit mode, the `+z` and `+Z` options are essentially the same. Normally, you compile with `+z`. However, in some instances — when the number of referenced symbols per shared library exceeds a predetermined limit — you must recompile with the `+Z` option instead. In this situation, the linker displays an error message and tells you to recompile the library with `+Z`.

In 64-bit mode, +Z is the default and the compilers ignore the options and generate PIC code.

Compiler Support for +z and +Z

In 32-bit mode, the C, C++, FORTRAN, and Pascal compilers support the `+z` and `+Z` options.

In 64-bit mode, `+Z` is the default for the C and C++ compilers.

Creating the Shared Library with ld

To create a shared library from one or more PIC object files, use the linker, `ld`, with the `-b` option. By default, `ld` will name the library `a.out`. You can change the name with the `-o` option.

For example, suppose you have three C source files containing routines to do length, volume, and mass unit conversions. They are named `length.c`, `volume.c`, and `mass.c`, respectively. To make a shared library from these source files, first compile all three files using the `+z` option, then combine the resulting `.o` files with `ld`. Shown below are the commands you would use to create a shared library named `libunits.sl`:

```
$ cc -Aa -c +z length.c volume.c mass.c
length.c:
volume.c:
mass.c:
$ ld -b -o libunits.sl length.o volume.o mass.o
```

Once the library is created, be sure it has read and execute permissions for all users who will use the library. For example, the following `chmod` command allows read/execute permission for all users of the `libunits.sl` library:

```
$ chmod +r+x libunits.sl
```

This library can now be linked with other programs. For example, if you have a C program named `convert.c` that calls routines from `libunits.sl`, you could compile and link it with the `cc` command:

```
$ cc -Aa convert.c libunits.sl
```

In 32-bit mode, once the executable is created, the library should not be moved because the absolute path name of the library is stored in the executable. (In 64-bit mode, `./libunit.sl` is stored in the executable.)For details, see “Shared Library Location”.

For details on linking shared libraries with your programs, see Chapter 3, “Linker Tasks,” on page 51.

NOTE

If you are linking any C++ object files to create an executable or a shared library, you must use the `CC` command to link. This ensures that `c++patch` executes and chains together your nonlocal static constructors and destructors. If you use `ld`, the library or executable may not work correctly and you will probably not get any error messages. For more information see the *HP C++ Programmer's Guide*.

Shared Library Dependencies

You can specify additional shared libraries on the `ld` command line when creating a shared library. The created shared library is said to have a **dependency** on the specified libraries, and these libraries are known as **dependent libraries** or **supporting libraries**. When you load a library with dependencies, all its dependent libraries are loaded too. For example, suppose you create a library named `libdep.sl` using the command:

```
$ ld -b -o libdep.sl mod1.o mod2.o -lcurses -lcustom
```

Thereafter, any programs that load `libdep.sl` — either explicitly with `shl_load` or implicitly with the dynamic loader when the program begins execution — also automatically load the dependent libraries `libcurses.sl` and `libcustom.sl`.

There are two additional issues that may be important to some shared library developers:

- When a shared library with dependencies is loaded, in what order are the dependent libraries loaded?
- Where are all the dependent libraries placed in relation to other already loaded libraries? That is, where are they placed in the process's shared library search list used by the dynamic loader?

The Order in Which Libraries Are Loaded (Load Graph)

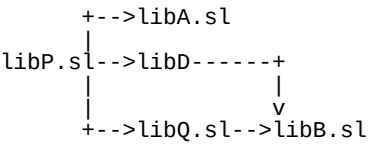
When a shared library with dependencies is loaded, the dynamic loader builds a load graph to determine the order in which the dependent libraries are loaded.

For example, suppose you create three libraries — `libQ`, `libD`, and `libP` — using the `ld` commands below. The order in which the libraries are built is important because a library must exist before you can specify it as a dependent library.

```
$ ld -b -o libQ.sl modq.o -lB
$ ld -b -o libD.sl modd.o -lQ -lB
$ ld -b -o libP.sl modp.o -lA -lD -lQ
```

The dependency lists for these three libraries are:

- `libQ` depends on `libB`
- `libD` depends on `libQ` and `libB`
- `libP` depends on `libA`, `libD`, and `libQ`



For 32-bit mode. The loader uses the following algorithm in 32-bit mode:

```
if the library has not been visited then
    mark the library as visited.
    if the library has a dependency list then
        traverse the list in reverse order.
    Place the library at the head of the load list.
```

Shown below are the steps taken to form the load graph when `libP` is loaded:

1. mark `P`, traverse `Q`
2. mark `Q`, traverse `B`
3. mark `B`, **load** `B`
4. **load** `Q`
5. traverse `D`
6. mark `D`, traverse `B`
7. `B` is already marked, so skip `B`, traverse `Q`
8. `Q` is already marked, so skip `Q`
9. **load** `D`
10. mark `A`, **load** `A`
11. **load** `P`

The resulting load graph is:

`libP` →`libA` →`libD` → `libQ` → `libB`

For 64-bit mode. The dynamic loader uses the following algorithm in 64-bit mode:

```
if the library has not been visited then
    mark the library as visited;
    append the library at the end of the list.
    if the library has a dependency list then
        traverse the list in reverse order.
```

Shown below are the steps taken to form the load graph when `libP` is loaded:

1. mark `P`, **load** `P`
2. traverse `P`

3. mark A, **load** A
4. mark D, **load** D
5. mark Q, **load** Q
6. traverse D
7. D is already marked, so skip D
8. traverse Q
9. Q is already marked, so skip Q
10. traverse Q
11. Q is already marked, so skip Q
12. traverse B
13. mark B, **load** B
14. traverse B
15. B is already marked, so skip B

The resulting load graph is:

libP →libA →libD → libQ → libB

Placing Loaded Libraries in the Search List

Once a load graph is formed, the libraries must be added to the shared library search list, thus binding their symbols to the program. If the initial library is an implicitly loaded library (that is, a library that is automatically loaded when the program begins execution), the libraries in the load graph are appended to the library search list. For example, if libP is implicitly loaded, the library search list is:

<current search list> → libP → libA → libD → libQ → libB

The same behavior occurs for libraries that are explicitly loaded with shl_load, but without the BIND_FIRST modifier (see "BIND_FIRST Modifier" on page 220 for details). If BIND_FIRST *is* specified in the shl_load call, then the libraries in the load graph are inserted *before* the existing search list. For example, suppose libP is loaded with this call:

```
lib_handle = shl_load("libP.sl", BIND_IMMEDIATE | BIND_FIRST, 0);
```

Then the resulting library search list is:

libP → libA → libD → libQ → libB → <*current search list*>

Updating a Shared Library

The ld command cannot replace or delete object modules in a shared library. Therefore, to update a shared library, you must relink the library with *all* the object files you want the library to include. For example, suppose you fix some routines in length.c (from the previous section) that were giving incorrect results. To update the libunits.sl library to include these changes, you would use this series of commands:

```
$ cc -Aa -c +z length.c
$ ld -b -o libunits.sl length.o volume.o mass.o
```

Any programs that use this library will now be using the new versions of the routines. That is, *you do not have to relink any programs that use this shared library*. This is because the routines in the library are attached to the program at run time.

This is one of the advantages of shared libraries over archive libraries: if you change an archive library, you must relink any programs that use the archive library. With shared libraries, you need only recreate the library.

Incompatible Changes to a Shared Library

If you make incompatible changes to a shared library, you can use library versioning to provide both the old and the new routines to ensure that programs linked with the old routines continue to work. See "Version Control with Shared Libraries" for more information on version control of shared libraries.

Shared Library Location

You can place shared libraries in the same locations as archive libraries (see "Archive Library Location"). Again, this is typically /usr/local/lib and /usr/contrib/lib (32-bit mode) or /user/local/lib/pa20_64 and /usr/contrib/lib/pa20_64 (64 bit mode) for application libraries, and /usr/lib (32-bit mode) or /user/lib/pa20_64 (64-bit mode) for system libraries. However, these are just suggestions.

Prior to the HP-UX 9.0 release, moving a shared library caused any programs that were linked with the library to fail when they tried to load the library. Prior to 9.0, you were required to relink all applications that used the library if the library was moved to a different directory.

Beginning with the HP-UX 9.0 release, a program can search a list of directories at run time for any required libraries. Thus, libraries can be moved after an application has been linked with them. To search for libraries at run time, a program must know which directories to search. There are two ways to specify this directory search information:

- Store a directory path list in the program with the linker option `+b` *path_list*.
- Link the program with `+s`, enabling the program to use the path list defined by the `SHLIB_PATH` environment variable at run time.

For 64-bit programs, you can also use the `LD_LIBRARY_PATH` environment variable, and the `+s` option is enabled by default.

For details on the use of these options, refer to “Moving Libraries after Linking with +b” on page 84 and “Moving Libraries After Linking with +s and SHLIB_PATH” on page 86.

Improving Shared Library Performance

This section describes methods you can use to improve the run-time performance of shared libraries. If, after using the methods described here, you are still not satisfied with the performance of your program with shared libraries, try linking with archive libraries instead to see if it improves performance. In general, though, archive libraries will not provide great performance improvements over shared libraries.

Using Profile-Based Optimization on Shared Libraries

You can perform profile-based optimization on your shared libraries to improve their performance. See “Profile-Based Optimization” on page 274 for more information.

Exporting Only the Required Symbols

Normally, all global variables and procedure definitions are exported from a shared library. In other words, any procedure or variable defined in a shared library is made visible to any code that uses this library. In addition, the compilers generate “internal” symbols that are exported.

You may be surprised to find that a shared library exports many more symbols than necessary for code that uses the library. These extra symbols add to the size of the library's symbol table and can even degrade performance (since the dynamic loader has to search a larger-than-necessary number of symbols).

One possible way to improve shared library performance is to export only those symbols that need exporting from a library. To control which symbols are exported, use either the `+e` or the `-h` option to the `ld` command. When `+e` options are specified, the linker exports only those symbols specified by `+e` options. The `-h` option causes the linker to hide the specified symbols. (For details on using these options, see "Hiding Symbols with -h" on page 81 and "Exporting Symbols with +e" on page 79.)

As an example, suppose you've created a shared library that defines the procedures `init_prog` and `quit_prog` and the global variable `prog_state`. To ensure that only these symbols are exported from the library, specify these options when creating the library:

```
+e init_prog +e quit_prog +e prog_state
```

If you have to export many symbols, you may find it convenient to use the `-c` *file* option, which allows you to specify linker options in *file*. For instance, you could specify the above options in a file named `export_opts` as:

```
+e init_prog
+e quit_prog
+e prog_state
```

Then you would specify the following option on the linker command line:

```
-c export_opts
```

(For details on the `-c` option, see "Passing Linker Options in a file with -c" on page 86.)

Placing Frequently-Called Routines Together

When the linker creates a shared library, it places the PIC object modules into the library in the order in which they are specified on the linker command line. The order in which the modules appear can greatly affect performance. For instance, consider the following modules:

`a.o`	Calls routines in `c.o` heavily, and its routines are called frequently by `c.o`.

b.o	A huge module, but contains only error routines that are seldom called.
c.o	Contains routines that are called frequently by a.o, and calls routines in a.o frequently.

If you create a shared library using the following command line, the modules will be inserted into the library in alphabetical order:

```
$ ld -b -o libabc.sl *.o
```

The potential problem with this ordering is that the routines in a.o and c.o are spaced far apart in the library. Better virtual memory performance could be attained by positioning the modules a.o and c.o together in the shared library, followed by the module b.o. The following command will do this:

```
$ ld -b -o libabc.sl a.o c.o b.o
```

One way to help determine the best order to specify the object files is to gather profile data for the object modules; modules that are frequently called should be grouped together on the command line.

Another way is to use the *lorder*(1) and *tsort*(1) commands. Used together on a set of object modules, these commands determine how to order the modules so that the linker only needs a single pass to resolve references among the modules. A side-effect of this is that modules that call each other may be positioned closer together than modules that don't. For instance, suppose you have defined the following object modules:

Module	Calls Routines in Module
a.o	x.o y.o
b.o	x.o y.o
d.o	none
e.o	none
x.o	d.o
y.o	d.o

Then the following command determines the one-pass link order:

```
$ lorder ?.o | tsort        Pipe lorder's output to tsort.
a.o
b.o
e.o
x.o
```

```
y.o
d.o
```

Notice that `d.o` is now closer to `x.o` and `y.o`, which call it. However, this is still not the best information to use because `a.o` and `b.o` are separated from `x.o` and `y.o` by the module `e.o`, which is *not* called by *any* modules. The actual optimal order might be more like this:

```
a.o b.o x.o y.o d.o e.o
```

Again, the use of `lorder` and `tsort` is not perfect, but it may give you leads on how to best order the modules. You may want to experiment to see what ordering gives the best performance.

Making Shared Libraries Non-Writable

You may get an additional performance gain by ensuring that no shared libraries have write permissions. Programs that use more than one writable library can experience significantly degraded loading time. The following `chmod` command gives shared libraries the correct permissions for best load-time performance:

```
$ chmod 555 libname
```

Using the +ESlit Option to cc

Normally, the C compiler places constant data in the data space. If such data is used in a shared library, each process will get its own copy of the data, in spite of the fact that the data is constant and should not change. This can result in some performance degradation.

To get around this, use the C compiler's `+ESlit` option, which places constant data in the `$LIT$` text space (or in 64-bit mode, in a `.text` text segment) instead of the data space. This results in one copy of the constant data being shared among all processes that use the library.

NOTE

This option requires that programs *not* write into constant strings and data. In addition, structures with embedded initialized pointers won't work because the pointers cannot be relocated since they are in read-only `$TEXT$` space. In this case, the linker outputs the error message "`Invalid loader fixup needed`".

Version Control with Shared Libraries

HP-UX provides two ways to support incompatible versions of shared library routines. “Library-Level Versioning” describes how you create multiple versions of a shared library. “Intra-Library Versioning” describes how a single shared library can have multiple versions of an object module. Library-level versioning is recommended over intra-library versioning.

NOTE Beginning with HP-UX Release 11.00, the 64-bit linker toolset supports only library-level versioning.

When to Use Shared Library Versioning

For the most part, updates to a shared library should be completely upward-compatible; that is, updating a shared library won't usually cause problems for programs that use the library. But sometimes — for example, if you add a new parameter to a routine — updates cause undesirable side-effects in programs that call the old version of the routine. In such cases, it is desirable to retain the old version as well as the new. This way, old programs will continue to run and new programs can use the new version of the routine.

Here are some guidelines to keep in mind when making changes to a library:

- When creating the first version of a shared library, carefully consider whether or not you will need versioning. It is easier to use library-level versioning from the start.

  When creating the first version of a shared library using intra-library versioning, version control is not an issue: The default version number is satisfactory.

- When creating future revisions of a library, you must determine when a change represents an incompatible change, and thus deserves a new version. Some examples of incompatible changes are as follows:

- As a general rule, when an exported function is changed such that calls to the function from previously compiled object files should not resolve to the new version, the change is *incompatible*. If the new version can be used as a wholesale replacement for the old version, the change is *compatible*.
- For exported data, any change in either the initial value or the size represents an incompatible change.
- Any function that is changed to take advantage of an incompatible change in another module should be considered incompatible.

Maintaining Old Versions of Library Modules

When using shared library versioning, you need to save the old versions of modules in the library:

- With library-level versioning, when an incompatible change is made to a module, the entire old version of the library must be retained along with the new version.
- With intra-library versioning, when an incompatible change is made to a module, all the old versions of the module should be retained along with the new version. The new version number should correspond to the date the change was made. If several modules are changed incompatibly in a library, it is a good idea to give all modules the same version date.

Library-Level Versioning

HP-UX 10.0 adds a new library-level versioning scheme that allows you to maintain multiple versions of shared libraries when you make incompatible changes to the library. By maintaining multiple versions, applications linked with the older versions continue to run with the older libraries, while new applications link and run with the newest version of the library. Library-level versioning is very similar to the library versioning on UNIX System V Release 4.

How to Use Library-Level Versioning

To use library-level versioning, follow these steps:

1. Name the first version of your shared library with an extension of `.0` (that's the number zero), for example `libA.0`. Use the `+h` option to designate the internal name of the library, for example, `libA.0`:

   ```
   ld -b *.o -o libA.0 +h libA.0        Creates the shared library libA.0.
   ```

2. Since the linker still looks for libraries ending in `.sl` with the `-l` option, create a symbolic link from the usual name of the library ending in `.sl` to the actual library. For example, `libA.sl` points to `libA.0`:

   ```
   ln -s libA.0 libA.sl                 libA.sl is a symbolic link to libA.0.
   ```

3. Link applications as usual, using the `-l` option to specify libraries. The linker searches for `libA.sl`, as usual. However, if the library it finds has an internal name, the linker places the internal name of the library in the executable's shared library dependency list. When you run the application, the dynamic loader loads the library named by this internal name. For example:

   ```
   ld /opt/langtools/lib/crt0.o prog.o -lA -lc  Binds a.out with libA.0.
   ```

Creating a New, Incompatible Version of the Library

When you create a new version of the library with incompatible changes, repeat the above steps except increment the number in the suffix of the shared library file name. That is, create `libA.1` rather than `libA.0` and set the symbolic link `libA.sl` to point to `libA.1`. Applications linked with `libA.0` continue to run with that library while new applications link and run with `libA.1`. Continue to increment the suffix number for subsequent incompatible versions, for example `libA.2`, `libA.3`, and so forth.

Migrating an Existing Library to Use Library-Level Versioning

If you have an existing library you can start using library-level versioning. First rename the existing library to have the extension `.0`. Then create the new version of the library with the extension `.1` and an internal name using the `.1` extension. Create a symbolic link with the extension `.sl` to point to the `.1` library.

When you run a program that uses shared libraries and was linked before HP-UX 10.0, the dynamic loader first attempts to open the shared library ending in `.0`. If it cannot find that library, it attempts to open the library ending in `.sl`.

The +h Option and Internal Names

The `+h` option gives a library an internal name for library-level versioning. Use `+h` to create versioned libraries:

`+h` *internal_name*

internal_name is typically the same name as the library file itself, for example `libA.1` as in `+h libA.1`. When you link with a library that has an internal name, the linker puts the *internal_name* in the shared library dependency list of the executable or shared library being created. When running the resulting executable or shared library, it is the library named by this internal name that the dynamic loader loads.

You can include a relative or absolute path with the internal name, but you must ensure the dynamic loader can find the library from this name using its normal search process.

For 32-bit mode, if *internal_name* includes a relative path (that is, a path not starting with `/`), the internal name stored by the linker is the concatenation of the actual path where the library was found and *internal_name*, including the relative path. When the resulting program is run, if the dynamic loader cannot find the library at this location, the program will not run.

If *internal_name* includes an absolute path (that is, a path starting with `/`), the internal name stored by the linker is simply the *internal_name*, including the absolute path. When the resulting program is run, if the dynamic loader cannot find the library at this location, the program will not run.

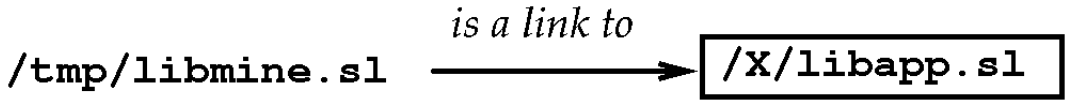
For 64-bit mode, see “Internal Name Processing” for more information.

File System Links to Shared Libraries

This section discusses the situation where you have used the *ln*(1) command to create file system links to shared library files. For example:

```
$ ld -b -o /X/libapp.sl *.o          Create shared library.
$ ln -s /X/libapp.sl /tmp/libmine.sl Make the link.
```

Figure 5-5



During a link, the linker records the file name of the opened library in the shared library list of the output file. However, if the shared library is a file system link to the actual library, the linker *does not* record the name of the library the file system link points to. Rather it records the name of the file system link.

For example, if `/tmp/libmine.sl` is a file system link to `/X/libapp.sl`, the following command records `/tmp/libmine.sl` in `a.out`, not `/X/libapp.sl` as might be expected:

```
$ ld /opt/langtools/lib/crt0.o main.o -L /tmp -lmine -lc
```

To use library-level versioning in this situation, you must set up corresponding file system links to make sure older applications linked with the older libraries run with these libraries. Otherwise, older applications could end up running with newer shared libraries. In addition, you must include the absolute path name in the internal name of the new library.

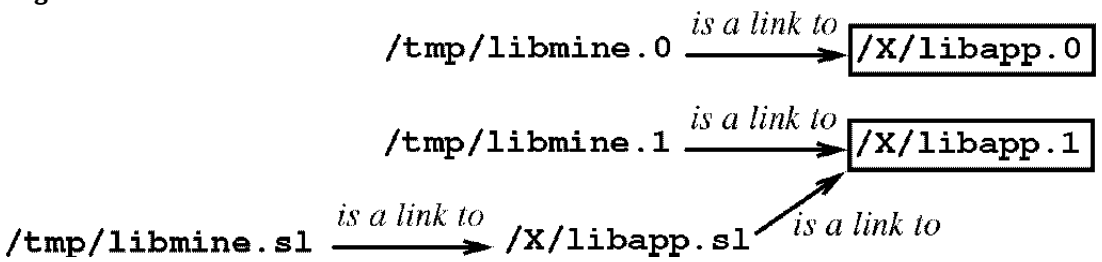
For example, in 32-bit mode, to make the above example work correctly with library-level versioning, first implement library-level versioning with the actual library `/X/libapp.sl` and include the absolute path in the internal name of the new library:

```
$ mv /X/libapp.sl /X/libapp.0                 Rename old version.
$ ld -b -o /X/libapp.1 +h /X/libapp.1 *.o     Create new version.
$ ln -s /X/libapp.1  /X/libapp.sl             Set up symbolic link.
```

Then set up the corresponding file system links:

```
$ ln -s /X/libapp.0 /tmp/libmine.0     Link to old version.
$ ln -s /X/libapp.1 /tmp/libmine.1     Link to new version.
$ rm /tmp/libmine.sl                   Remove old link.
$ ln -s /X/libapp.sl /tmp/libmine.sl   Link to the link.
```

Figure 5-6



With these links in place, the loader will load `/X/libapp.0` when running the `a.out` file created above. New applications will link and run with `/X/libapp.1`.

NOTE Renaming the old version of the `.0` version library only works for 32-bit mode.

For 64-bit mode programs, the dynamic loader only loads the library recorded in the dynamic load table. You should use library-lever versioning and create your 64-bit shared library with an internal name unless the library will not be versioned in the future.

Using shl_load(3X) with Library-Level Versioning

Once library level versioning is used, calls to *shl_load*(3X) should specify the actual version of the library to be loaded. For example, if `libA.sl` is now a symbolic link to `libA.1`, then calls to dynamically load this library should specify the latest version available when the application is compiled as shown below:

```
shl_load("libA.1", BIND_DEFERRED, 0);
```

This insures that when the application is migrated to a system that has a later version of `libA` available, the actual version desired is the one that is dynamically loaded.

Intra-Library Versioning

Intra-library versioning is a second method of maintaining multiple incompatible versions of shared library routines. Library-level versioning is recommended over intra-library versioning.

This section provides information on the following topics:

- "The Version Number Compiler Directive"

- "Shared Library Dependencies and Version Control"
- "Adding New Versions to a Shared Library"
- "Specifying a Version Date"

The Version Number Compiler Directive

With intra-library versioning, you assign a **version number** to any module in a shared library. The version number applies to all global symbols defined in the module's source file. The version number is a date, specified with a compiler directive in the source file. The syntax of the version number directive depends on the language:

C and C++: `#pragma HP_SHLIB_VERSION "`*date*`"`

FORTRAN: `$SHLIB_VERSION '`*date*`'`

Pascal: `$SHLIB_VERSION '`*date*`'$`

The *date* argument in all three directives is of the form *month*/*year*. The *month* must be `1` through `12`, corresponding to January through December. The *year* can be specified as either the last two digits of the year (`94` for 1994) or a full year specification (`1994`). Two-digit year codes from `00` through `40` represent the years 2000 through 2040.

This directive should only be used if incompatible changes are made to a source file. If a version number directive is not present in a source file, the version number of all symbols defined in the object module defaults to `1/90`.

Shared Library Dependencies and Version Control

A shared library as a whole can be thought of as having a version number itself. This version number is the most recent of the versioned symbols in the library and any dependent libraries.

When a shared library is built with a dependent shared library, the version number of the dependent library used during the link is recorded with the dependency.

When *shl_load*(3X) is called to load a shared library, the latest version of the library is loaded. If that library has dependencies, the dynamic loader (*dld.sl*(5)) will load the versions of the dependent libraries that were recorded in the dependency list. Note that these are not necessarily the most recent versions of the dependent libraries. When `dld.sl` loads a particular version of a shared library, it will load the same version of any dependent libraries.

If a shared library lists a second shared library as a dependency, `dld.sl` will generate an error if the second shared library has a version number which is older than the version number recorded with the dependency. This means that the first library was built using a more recent version of the second library than the version that `dld.sl` currently finds.

Adding New Versions to a Shared Library

To rebuild a shared library with *new* versions of object files, run `ld` again with the newly compiled object files. For example, suppose you want to add new functionality to the routines in `length.c`, making them incompatible with existing programs that call `libunits.sl`. Before making the changes, make a copy of the existing `length.c` and name it `oldlength.c`. Then change the routines in `length.c` with the version directive specifying the current month and date. The following shows the new `length.c` file:

```
#pragma HP_SHLIB_VERSION "11/93" /* date is November 1993 */
/*
 * New version of "in_to_cm" also returns a character string
 * "cmstr" with the converted value in ASCII form.
 */
float   in_to_cm(float in, char *cmstr)     /* convert in to cm *
/
{
       . .  .          /* build "cmstr" */
        return(in * 2.54);
}
       . . .           /* other length conversion routines */
```

To update `libunits.sl` to include the new `length.c` routines, copy the old version of `length.o` to `oldlength.o`; then compile `length.c` and rebuild the library with the new `length.o` and `oldlength.o`:

```
$ cp length.c oldlength.c              Save the old source.
$ mv length.o oldlength.o              Save old length.o.
          . . .                        Make new length.c.
$ cc -Aa -c +z length.c                Make new length.o.
$ ld -b -o libunits.sl oldlength.o \   Relink the library.

    volume.o mass.o length.o
```

Thereafter, any programs linked with `libunits.sl` use the new versions of length-conversion routines defined in `length.o`. Programs linked with the old version of the library still use those routines from `oldlength.o`. For details on linking with shared libraries, see Chapter 3, “Linker Tasks,” on page 51.

Specifying a Version Date

When adding modules to a library for a particular release of the library, it is best to give all modules the same version date. For example, if you complete `file1.o` on 04/93, `file2.o` on 05/93, and `file3.o` on 07/93, it would be best to give all the modules the same version date, say 07/93.

The reason for doing this is best illustrated with an example. Suppose in the previous example you gave each module a version date corresponding to the date it was completed: 04/93 for `file1.o`, 05/93 for `file2.o`, and 07/93 for `file3.o`. You then build the final library on 07/93 and link an application `a.out` with the library. Now suppose that you introduce an incompatible change to function `foo` found in `file1.o`, set the version date to 05/93, and rebuild the library. If you run `a.out` with the new version of the library, `a.out` will get the new, incompatible version of `foo` because its version date is still earlier than the date the application was linked with the original library!

Switching from Archive to Shared Libraries

There are cases where a program may behave differently when linked with shared libraries than when linked with archive libraries. These are the result of subtle differences in the algorithms the linker uses to resolve symbols and combine object modules. This section covers these considerations. (See also "Caution When Mixing Shared and Archive Libraries".)

Library Path Names

As discussed previously, in 32-bit mode, `ld` records the absolute path names of any shared libraries searched at link time in the `a.out` file. When the program begins execution, the dynamic loader attaches any shared libraries that were specified at link time. Therefore, you must ensure that any libraries specified at link time are also present in the same location at run time.

For 64-bit naming, see "Internal Name Processing" for more information.

Relying on Undocumented Linker Behavior

Occasionally, programmers may take advantage of linker behavior that is undocumented but has traditionally worked. With shared libraries, such programming practices might not work or may produce different results. If the old behavior is absolutely necessary, linking with archive libraries only (`-a archive`) produces the old behavior.

For example, suppose several definitions and references of a symbol exist in different object and archive library files. By specifying the files in a particular link order, you could cause the linker to use one definition over another. But doing so requires an understanding of the subtle (and undocumented) symbol resolution rules used by the linker, and these rules are slightly different for shared libraries. So `make` files or shell scripts that took advantage of such linker behavior prior to the support of shared libraries may not work as expected with shared libraries.

More commonly, programmers may take advantage of undocumented linker behavior to minimize the size of routines copied into the `a.out` files from archive libraries. This is no longer necessary if all libraries are shared.

Although it is impossible to characterize the new resolution rules exactly, the following rules always apply:

- If a symbol is defined in two shared libraries, the definition used at run time is the one that appeared first, regardless of where the reference was.
- The linker treats shared libraries more like object files.

As a consequence of the second rule, programs that call wrapper libraries may become larger. (A **wrapper library** is a library that contains alternate versions of C library functions, each of which performs some bookkeeping and then calls the actual C function. For example, each function in the wrapper library might update a counter of how many times the actual C routine is called.) With archive libraries, if the program references only one routine in the wrapper library, then only the wrapper routine and the corresponding routine from the C library are copied into the `a.out` file. If, on the other hand, a shared wrapper library and archive C library are specified, in that order, then *all routines that can be referenced by any routine in the wrapper library are copied from the C library*. To avoid this, link with archive or shared versions for both the wrapper library and C library, or use an archive version of the wrapper library and a shared version of the C library.

Absolute Virtual Addresses

Writing code that relies on the linker to locate a symbol in a particular location or in a particular order in relation to other symbols is known as making an **implicit address dependency**. Because of the nature of shared libraries, the linker cannot always preserve the exact ordering of symbols declared in shared libraries. In particular, variables declared in a shared library may be located far from the main program's virtual address space, and they may not reside in the same relative order within the library as they were linked. Therefore, code that has implicit address dependencies may not work as expected with shared libraries.

An example of an implicit address dependency is a function that assumes that two global variables that were defined adjacently in the source code will actually be adjacent in virtual memory. Since the linker may

rearrange data in shared libraries, this is no longer guaranteed. Another example is a function that assumes variables it declares statically (for example, C `static` variables) reside below the reserved symbol `_end` in memory (see *end*(3)). In general, it is a bad idea to depend on the relative addresses of global variables, because the linker may move them around.

In assembly language, using the address of a label to calculate the size of the immediately preceding data structure is not affected: the assemblers still calculate the size correctly.

Stack Usage

To load shared libraries, a program must have a copy of the dynamic loader (`dld.sl`) mapped into its address space. This copy of the dynamic loader shares the stack with the program. The dynamic loader uses the stack during startup and whenever a program calls a shared library routine for the first time. If you specify `-B immediate`, the dynamic loader uses the stack at startup only.

NOTE

For 32-bit mode only:

Although it is not recommended programming practice, some programs may use stack space “above” the program's current stack. To preserve the contents “above” the program's logical top of the stack, the dynamic loader attempts to use stack space far away from program's stack pointer. If a program is doing its own stack manipulations, such as those implemented by a “threads” package, the dynamic loader may inadvertently use stack space that the program had reserved for another thread. Programs doing such stack manipulations should link with archive libraries, or at least use immediate binding, if this could potentially cause problems.

Also be aware that if a program sets its stack pointer to memory allocated in the heap, the dynamic loader may use the space directly “above” the top of this stack when deferred binding of symbols is used.

Version Control

You can maintain multiple versions of a shared library using library-level versioning. This allows you to make incompatible changes to shared libraries and ensure programs linked with the older versions continue to run. (See “Library-Level Versioning” for more information.)

Debugger Limitations

Shared libraries can be debugged just like archive libraries with few exceptions. For details on debugging shared libraries, refer to the *HP/DDE Debugger User's Guide* or the *HP-UX Symbolic Debugger User's Guide*.

Using the chroot Command with Shared Libraries

Some users may use the `chroot` super-user command when developing and using shared libraries. This affects the path name that the linker stores in the executable file. For example, if you `chroot` to the directory `/users/hyperturbo` and develop an application there that uses the shared library `libhype.sl` in the same directory, `ld` records the path name of the library as `/libhype.sl`. If you then exit from the `chroot`ed directory and attempt to run the application, the dynamic loader won't find the shared library because it is actually stored in `/users/hyperturbo/libhype.sl`, not in `/libhype.sl`.

Conversely, if you move a program that uses shared libraries into a `chroot`ed environment, you must have a copy of the dynamic loader, `dld.sl`, and all required shared libraries in the correct locations.

Profiling Limitations

Profiling with the `prof`(1) and `gprof`(1) commands and the `monitor` library function is only possible on a contiguous chunk of the main program (`a.out`). Since shared libraries are not contiguous with the main program in virtual memory, they cannot be profiled. You can still profile the main program, though. If profiling of libraries is required, relink the application with the archive version of the library, using the `-a archive` option.

Summary of HP-UX Libraries

What libraries your system has depends on what components were purchased. For example, if you didn't purchase Starbase Display List, you won't have the Starbase Display List library on your system.

HP-UX library routines are described in detail in sections 2 and 3 of the *HP-UX Reference*. Routines in section 2 are known as **system calls**, because they provide low-level system services; they are found in `libc`. Routines in section 3 are other "higher-level" library routines and are found in several different libraries including `libc`.

Each library routine, or group of library routines, is documented on a **man page**. Man pages are sorted alphabetically by routine name and have the general form *routine*(*nL*), where:

routine	is the name of the routine, or group of closely related routines, being documented.
n	is the *HP-UX Reference* section number: 2 for system calls, 3 for other library routines.
L	is a letter designating the library in which the routine is stored.

For example, the *printf*(3S) man page describes the standard input/output `libc` routines `printf`, `nl_printf`, `fprintf`, `nl_fprintf`, `sprintf`, and `nl_sprintf`. And the *pipe*(2) man page describes the `pipe` system call.

The major library groups defined in the *HP-UX Reference* are shown below:

NOTE

Certain language-specific libraries are not documented in the *HP-UX Reference*; instead, they are documented with the appropriate language documentation. For example, all FORTRAN intrinsics (`MAX`, `MOD`, and so forth) are documented in the *HP FORTRAN/9000 Programmer's Reference*.

Group	Description

(2)	These functions are known as system calls. They provide low-level access to operating system services, such as opening files, setting up signal handlers, and process control. These routines are located in `libc`.
(3C)	These are standard *C* library routines located in `libc`.
(3S)	These functions comprise the *S*tandard input/output routines (see *stdio*(3S)). They are located in `libc`.
(3M)	These functions comprise the *M*ath library. The linker searches this library under the `-lm` option (for the SVID math library) or the `-lM` option (for the POSIX math library).
(3G)	These functions comprise the *G*raphics library.
(3I)	These functions comprise the *I*nstrument support library.
(3X)	Various specialized libraries. The names of the libraries in which these routines reside are documented on the man page.

The routines marked by (2), (3C), and (3S) comprise the standard C library `libc`. The C, C++, FORTRAN, and Pascal compilers automatically link with this library when creating an executable program.

For more information on these libraries, see *C, A Reference Manual* by Samual P. Harbison and Guy L. Steele Jr., published in 1991 by Prentice-Hall, or *UNIX System V Libraries* by Baird Peterson, published in 1992 by Van Nostrand Reinhold, or *C Programming for UNIX* by John Valley, published in 1992 by Sams Publishing. For more information on system calls see *Advanced UNIX Programming* by Marc J. Rochkind, published in 1985 by Prentice-Hall or *Advanced Programming in the UNIX Environment* by W. Richard Stevens, published in 1992 by Addison-Wesley.

Caution When Mixing Shared and Archive Libraries

Mixing shared and archive libraries in an application is not recommended and should be avoided. That is, an application should use *only* shared libraries or *only* archive libraries.

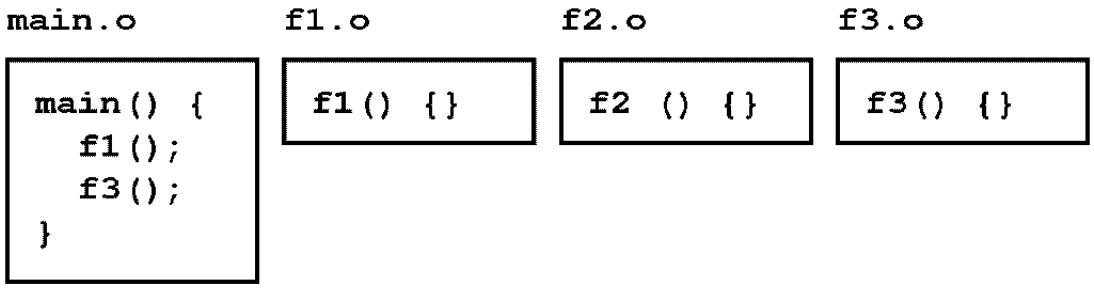
Mixing shared and archive libraries can lead to unsatisfied symbols, hidden definitions, and duplicate definitions and cause an application to abort or exhibit incorrect behavior at run time. The following examples illustrate some of these problems.

Example 1: Unsatisfied Symbols

This example (in 32-bit and 64-bit `+compat` mode) shows how mixing shared and archive libraries can cause a program to abort. Suppose you have a main program, `main()`, and three functions, `f1()`, `f2()`, and `f3()` each in a separate source file. `main()` calls `f1()` and `f3()` but not `f2()`:

```
$ cc -c main.c f1.c f2.c      Compile to relocatable object code.
$ cc -c +z f3.c               Compile to position-independent code
```

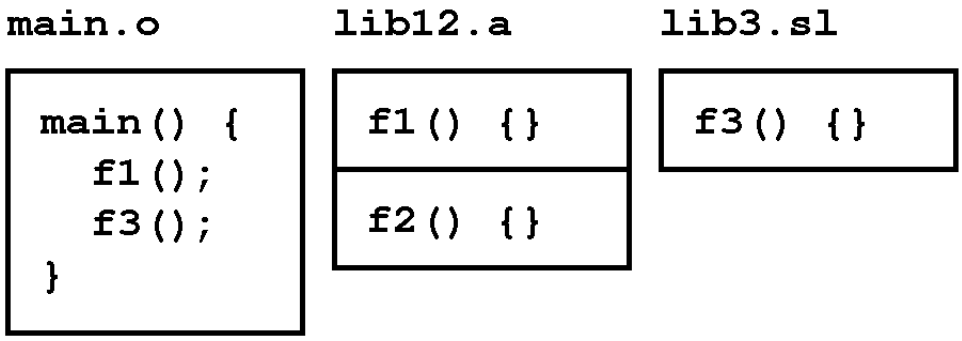
Figure 5-7



Next suppose you put `f3.o` into the shared library `lib3.sl` and `f1.o` and `f2.o` into the archive library `lib12.a`:

```
$ ld -b -o lib3.sl f3.o       Create a shared library.
$ ar qvc lib12.a f1.o f2.o    Create an archive library.
```

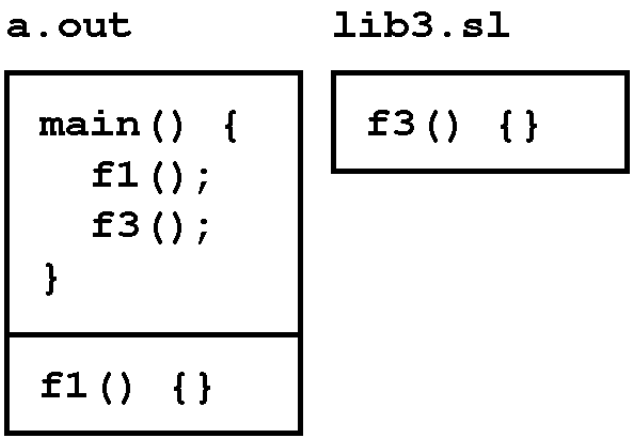
Figure 5-8



Now link the main with the libraries and create the executable `a.out`:

```
$ cc main.o lib12.a lib3.sl      Link the program
```

Figure 5-9



When you run `a.out`, it runs correctly. Now suppose you need to modify `f3()` to call `f2()`:

Figure 5-10

f3.c

```
f3() {
 f2();
}
```

Compile the new `f3()` and rebuild the shared library `lib3.sl`:

```
$ cc -c +z f3.c            Compile to relocatable code
.$ ld -b -o lib3.sl f3.o   Create a new shared library.
```

Figure 5-11

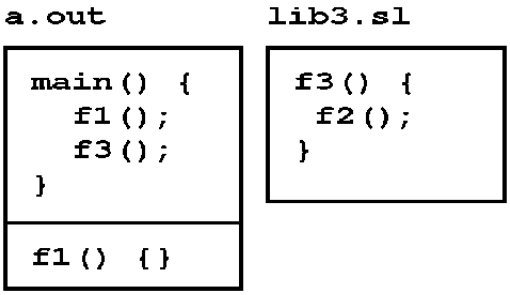
lib3.sl

```
f3() {
 f2();
}
```

Problem

Here's where the problem can occur. If you *do not* relink the application, `main.o`, and just run `a.out` with the new version of `lib3.sl`, the program will abort since `f2()` is not available in the application. The reference to `f2()` from `f3()` remains unsatisfied, producing an error in 32-bit mode:

Figure 5-12




```
$ a.out
/usr/lib/dld.sl: Unresolved symbol: f2 (code) from
                 /users/steve/dev/lib3.sl
Abort(coredump)
```

Example 2: Using shl_load(3X)

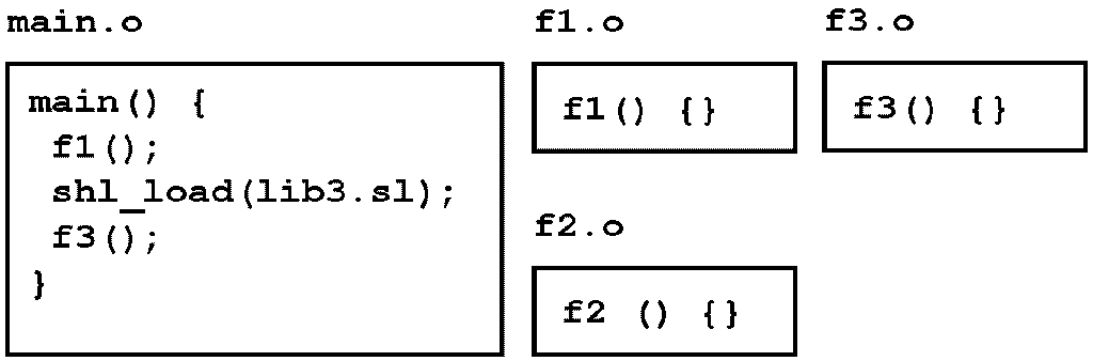
This example (in 32-bit and 64-bit `+compat` mode shows how mixing archive libraries and shared libraries using *shl_load*(3X) can lead to unsatisfied symbols and cause a program to abort.

If a library being loaded depends on a definition that does not exist in the application or any of the dependent shared libraries, the application will abort with an unsatisfied definition at run time. This seems obvious enough when an application is first created. However, over time, as the shared libraries evolve, new symbol imports may be introduced that were not originally anticipated. This problem can be avoided by ensuring that shared libraries maintain accurate dependency lists.

Suppose you have a main program, `main()`, and three functions, `f1()`, `f2()`, and `f3()` each in a separate source file. `main()` calls `f1()` and uses `shl_load()` to call `f3()`. `main()` does not call `f2()`:

```
$ cc -c main.c f1.c f2.c  Compile to relocatable object code
$ cc -c +z f3.c           Compile to position-independent code
```

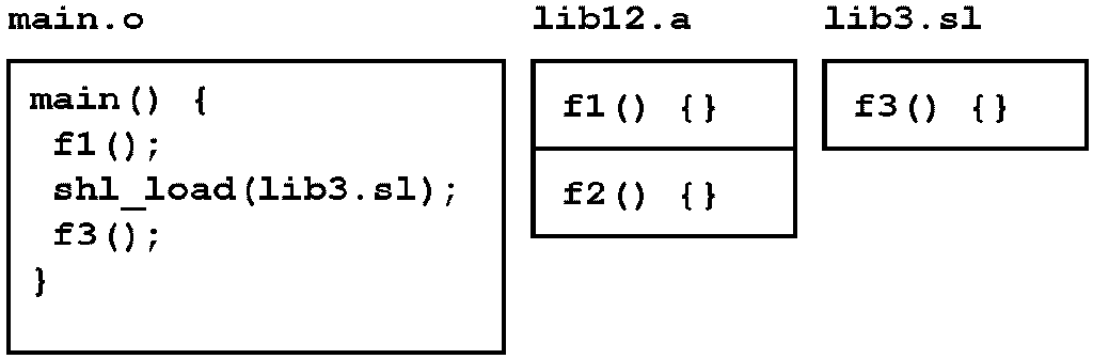
Figure 5-13



Next suppose you put `f3.o` into the shared library `lib3.sl` and `f1.o` and `f2.o` into the archive library `lib12.a`:

```
$ ld -b -o lib3.sl f3.o       Create a shared library.
$ ar qvc lib12.a f1.o f2.o    Create an archive library.
```

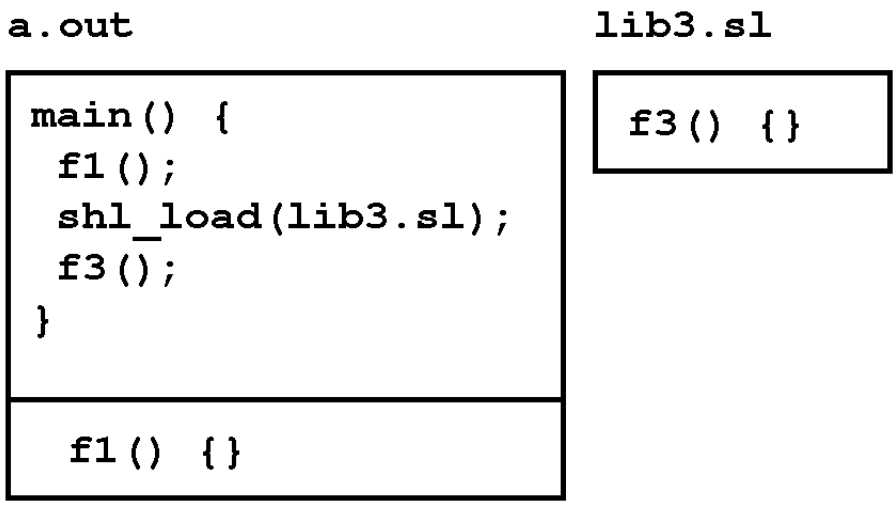
Figure 5-14



Now link the main with the archive library and create the executable `a.out`:

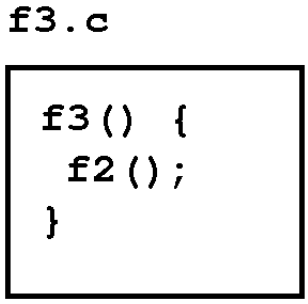
```
$ cc main.o lib12.a -ldld       Link the program.
```

Figure 5-15



When you run `a.out`, it runs correctly. Now suppose you need to modify `f3()` to call `f2()`:

Figure 5-16

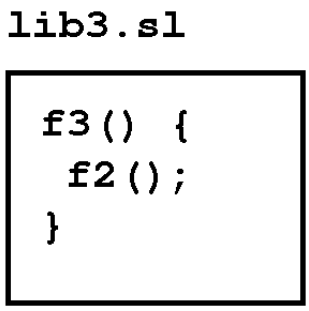


Problem

Here is where a problem can be introduced. If you compile the new `f3()` and rebuild the shared library `lib3.sl` *without specifying the dependency on a shared library containing* `f2()`, calls to `f3()` will abort.

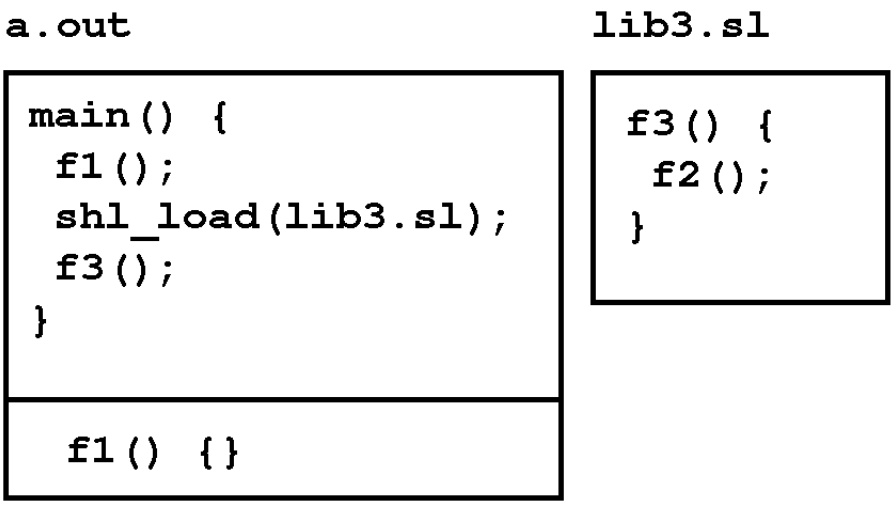
```
$ cc -c +z f3.c              Compile to position-independent code.
$ ld -b -o lib3.sl f3.o      Error! Missing library containing f2().
```

Figure 5-17



Here's where the problem shows up. If you *do not* relink the application, `main.o`, and just run `a.out` with the new version of `lib3.sl`, the program will abort since `f2()` is not available in the program's address space. The reference to `f2()` from `f3()` remains unsatisfied, generating the 32-bit error message:

Figure 5-18




```
$ a.out
Illegal instruction (coredump)
```

Example 3: Hidden Definitions

This example shows how mixing archive libraries and shared libraries can lead to multiple definitions in the application and unexpected results. If one of the definitions happens to be a data symbol, the results can be catastrophic. If any of the definitions are code symbols, different versions of the same routine could end up being used in the application. This could lead to incompatibilities.

Duplicate definitions can occur when a dependent shared library is updated to refer to a symbol contained in the program file but not visible to the shared library. The new symbol import must be satisfied somehow by either adding the symbol to the library or by updating the shared library dependency list. Otherwise the application must be relinked.

Using an archive version of `libc` in an application using shared libraries is the most common cause of duplicate definitions. Remember that symbols not referenced by a shared library at link time will not be exported by default.

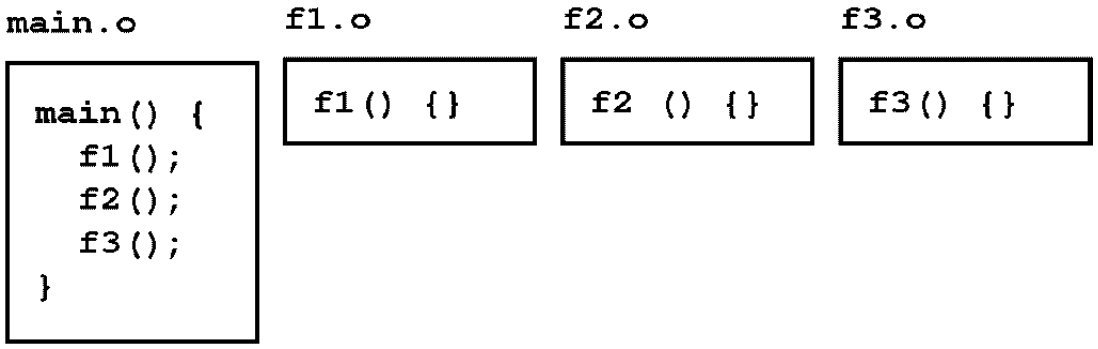
NOTE

Duplicate definitions can be avoided if any or all symbols that may be referenced by a shared library are exported from the application at link time. Shared libraries always reference the first occurrence of a definition. In the following example the first definition is in the executable file, `a.out`. See the `-E` option and `+e` *symbol* option described in *ld*(1) and "Exporting Symbols from main with -E" on page 81, "Exporting Symbols with +ee" on page 81, and "Exporting Symbols with +e" on page 79.

The following example illustrates this situation. Suppose you have a main program, `main()`, and three functions, `f1()`, `f2()`, and `f3()` each in a separate source file. `main()` calls `f1()`, `f2()`, and `f3()`.

```
$ cc -c main.c              Compile to relocatable code.
$ cc -c +z f1.c f2.c f3.c   Compile to position-independent code.
```

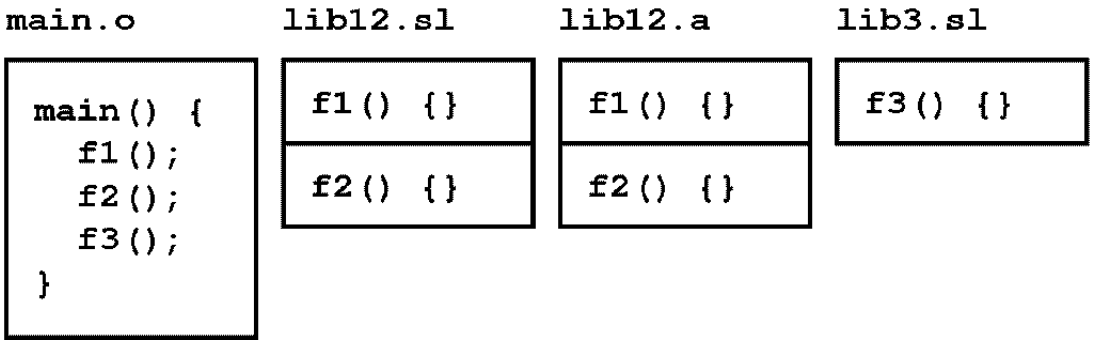
Figure 5-19



Next suppose you put `f3.o` into the shared library `lib3.sl` and `f1.o` and `f2.o` into the archive library `lib12.a`. Also put `f1.o` and `f2.o` into the shared library `lib12.sl`:

```
$ ld -b -o lib3.sl f3.o          Create a shared library.
$ ld -b -o lib12.sl f1.o f2.o    Create a shared library.
$ ar qvc lib12.a f1.o f2.o       Create an archive library.
```

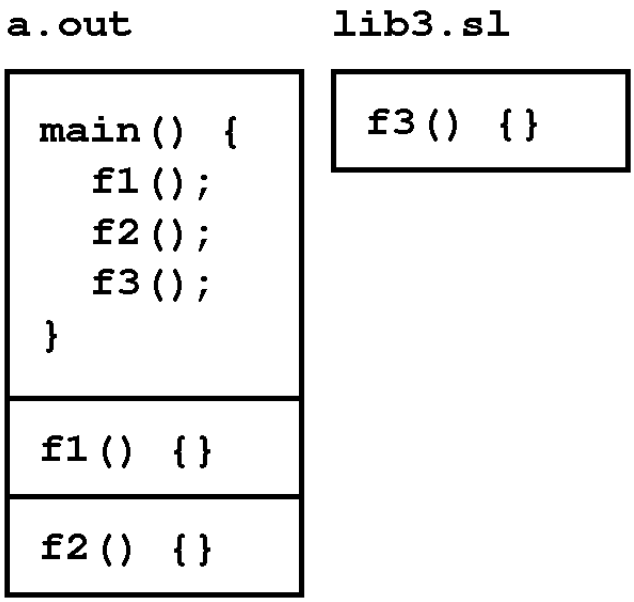
Figure 5-20



Now link the main with the archive library `lib12.a` and the shared library `lib3.sl` and create the executable `a.out`:

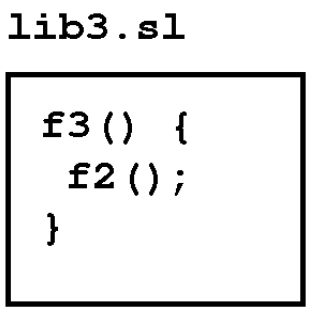
```
$ cc main.o lib12.a lib3.sl      Link the program.
```

Figure 5-21



When you run `a.out`, it runs correctly. Now suppose you need to modify `f3()` to call `f2()`:

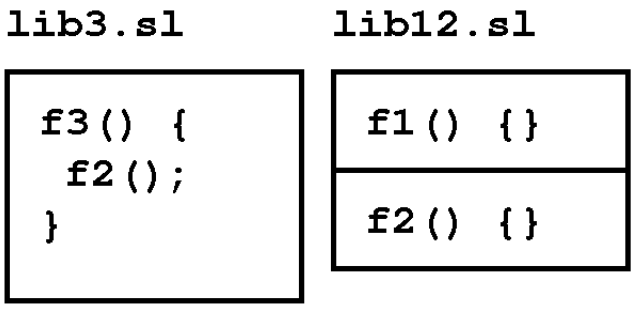
Figure 5-22



Compile the new `f3()` and rebuild the shared library `lib3.sl`, including the new dependency on `f2()` in `lib12.sl`:

```
$ cc -c +z f3.c                    Compile to PIC.
$ ld -b -o lib3.sl f3.o -L . -l12  Create library with dependency.
```

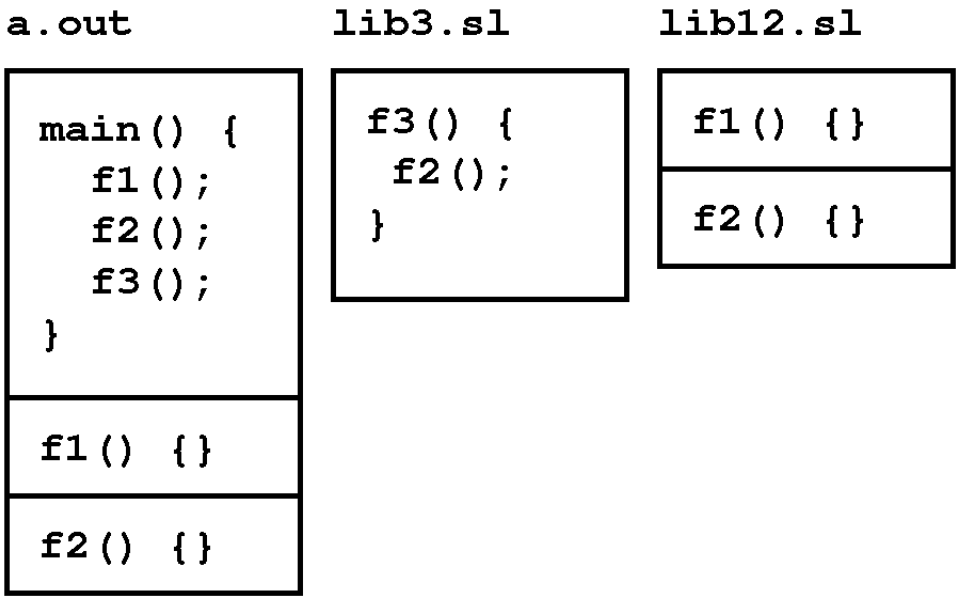
Figure 5-23



Problem

Here's where the problem can occur in 32-bit and 64-bit `+compat` modes. If you *do not* relink the application, `main.o`, and just run `a.out` with the new version of `lib3.sl`, the program will execute successfully, but it will execute *two different versions* of `f2()`. `main()` calls `f2()` in the program file `a.out` and `f3()` calls `f2()` in `lib12.sl`. Even though `f2()` is contained within the application, it is not visible to the shared library, `lib3.sl`.

Figure 5-24



Summary of Mixing Shared and Archive Libraries

Applications that depend on shared libraries should not use archive libraries to satisfy symbol imports in a shared library. This suggests that only a shared version of `libc` should be used in applications using shared libraries. If an archive version of a dependent library must be used, all of its definitions should be explicitly exported with the `-E` or `+e` options to the linker to avoid multiple definitions.

Providers of shared libraries should make every effort to prevent these kinds of problems. In particular, if a shared library provider allows unsatisfied symbols to be satisfied by an archive version of `libc`, the application that uses the library may fail if the shared library is later updated and any new `libc` dependencies are introduced. New dependencies in shared libraries can be satisfied by maintaining accurate dependency lists. However, this can lead to multiple occurrences of the same definition in an application if the definitions are not explicitly exported.

Using Shared Libraries in 64-bit mode

In the HP-UX 11.00 release, HP provides an industry-standard linker toolset for programs linked in 64-bit mode. The new toolset consists of a linker, dynamic loader, object file class library, and an object file tool collection. Although compatibility between the current and previous toolset is an important goal, some differences exist between these toolsets.

The 64-bit mode linker toolset introduces different types of shared libraries. (In SVR4 Unix, shared libraries are sometimes called **dlls**.)

- **Compatibility mode shared library**: Using the 64-bit mode linker, a compatibility mode shared library is basically a library built with `ld -b +compat` that has dependent shared libraries. The `+compat` option affects the way the linker and loader search for dependent libraries of a shared library and records their names.
- **Standard mode shared library**: A standard mode shared library is a library built with `ld -b` or `ld -b +std` with dependent shared libraries.

NOTE If you specify `ld -b +compat` with no dependent libraries, you create a shared library that has no mode —neither compatibility mode nor standard mode.

The linker handles these libraries in different way with regard to internal naming and library search modes.

Internal Name Processing

For both 32-bit mode and 64-bit mode, you specify shared library internal names using `ld +h` *name* However, their dependent libraries' internal names may not be recorded the same way in a standard mode link.

In an `ld +compat` link, the linker treats the internal names like it does in 32-bit mode:

- If the dependent library's internal name is rooted (starts with "/"), it is inserted as is in the `DT_HP_NEEDED` entry. If it was specified with `-l`, the dynamic path bit is set to `TRUE` in the `DT_HP_NEEDED` entry.

- If the dependent library's internal name contains a relative path, the internal name is inserted at the end of the path where the shared library is found at link time, replacing the library's filename in the DT_HP_NEEDED entry. If the library is specified with -l, the dynamic path bit is set to TRUE.
- If the dependent library's internal name contains no path, it is inserted at the end of the path where the shared library is found at link time, replacing the library's filename. If the library is specified with -l, the dynamic path bit is set to TRUE.
- If the dependent shared library does not have an internal name, the path where the library is found and the library filename is recorded in the DT_HP_NEEDED entry. If specified with -l, the dynamic path bit is set to TRUE.
- If the shared libraries are specified with a relative or no path in this mode, the linker expands the current working directory when constructing the DT_HP_NEEDED entry. So instead of getting something like ./libdk.sl, you get /home/your_dir/libdk.sl.
- All DT_HP_NEEDED entries with the dynamic path bit set are subject to dynamic path lookup.

In standard mode, the linker treats shared library names as follows:

- If the dependent shared library has an internal name, it is recorded in the DT_NEEDED entry.

  otherwise

- If the dependent shared library is specified with the -l or -l: option, only the *libname.ext* is recorded in the DT_NEEDED entry.

  otherwise

- The path of the dependent shared library as seen on the link line is recorded in the DT_NEEDED entry.
- All DT_NEEDED entries with no "/" in the libname are subject to dynamic path lookup.

Dynamic Path Searching for Shared Libraries

In the 64-bit mode of the linker toolset (selected by default or with the +std option), any library whose name has no "/" character in it becomes a candidate for dynamic path searching. Also, the linker always uses the

LD_LIBRARY_PATH and the SHLIB_PATH environment variable to add directories to the run time search path for shared libraries, unless the ld +noenvvar option is set.

In the 32-bit mode of the linker toolset (selected by the +compat option), the linker enables run-time dynamic path searching when you link a program with the -l*library* and +b *path_name* options. Or, you can use the -l*library* option with the +s *path_name* option. When programs are linked with +s, the dynamic loader searches directories specified by the SHLIB_PATH environment variable to find shared libraries.

The following example shows dynamic path searching changes for 64-bit mode.

```
ld /opt/langtools/lib/crt0.o main.o \      Subject to
 -lfoo -o main                             dynamic path searching.
```

In 32-bit mode, main aborts at run time if libfoo.sl is moved from its standard location, /usr/lib or /usr/ccs/lib. The linker does not do dynamic path searching unless you specify the +b or +s options to the ld or chatr commands. In 64-bit mode, the dynamic loader searches for libfoo.sl in the directories specified by the LD_LIBRARY_PATH and SHLIB_PATH environment variables.

Shared Library Symbol Binding Semantics

Symbol binding resolution, both at link time and run time, changes slightly in the 64-bit mode HP-UX linker toolset. The symbol binding policy is more compatible with other open systems.

This section covers the following topics:

- Link-time symbol resolution in shared libraries
- Resolution of unsatisfied shared library references
- Promotion of uninitialized global variables
- Symbol searching in dependent libraries

Link-Time Symbol Resolution in Shared Libraries

In the 64-bit mode of the linker toolset, the linker remembers all symbols in a shared library for the duration of the link. Global definitions satisfy trailing references, and unsatisfied symbols remaining in shared libraries are reported.

The 32-bit mode linker does not remember the definition of a procedure in a shared library unless it was referenced in previously scanned object files.

If you have function names that are duplicated in a shared and archive library, the 64-bit mode linker may reference a different version of a procedure than is referenced by the 32-bit mode linker. This change can lead to unexpected results.

For example, given these source files:

sharedlib.c

```
void afunc()
{
    printf("\tin SHARED library procedure 'afunc'\n");
}
```

unsat.c

```
void bfunc()
{
     afunc();
}
```

archive.c

```
void afunc()
{
    printf ("\tin ARCHIVE library procedure 'afunc'\n");
}
```

main.c

```
main()
{
    bfunc();
}
```

If these files are compiled and linked as:

```
cc -c main.c unsat.c archive.c
cc -c +z sharedlib.c
ld -b sharedlib.o -o libA.sl
ar rv libB.a archive.o
cc main.o libA.sl unsat.o libB.a -o test1
```

The 32-bit linker toolset produces:

```
$ test1

        in ARCHIVE library procedure `afunc'
```

At link time, there is an outstanding unsatisfied symbol for `afunc()` when `libB` is found. The exported symbol for `afunc()` *is not remembered* after `libA.sl` is scanned. At run time, the `afunc()` symbol that is called is the one that came from the archive library, which resides in `test1`.

The 64-bit mode linker toolset produces:

```
$ test1
        in SHARED library procedure `afunc'
```

The 64-bit mode linker *remembers* the symbol for `afunc()`, and `archive.o` will not be pulled out of `libB.a`. The shared library version of `afunc` is called during execution. This behavior is consistent with other SVR4 systems.

Resolution of Unsatisfied Shared Library References

In the 64-bit mode linker toolset, the dynamic loader requires that all symbols referenced by all loaded libraries be satisfied at the appropriate time. This is consistent with other SVR4 systems.

The 32-bit mode linker toolset accepts unresolved symbols in some cases. For example, if an entry point defined in an object file is never reachable from the main program file, the unresolved symbol is allowed. You can use the `+vshlibunsats` linker option to find unresolved symbols in shared libraries.

For example, given these source files:

`lib1.c`

```
void a()
{
}
```

`lib2.c`

```
extern int unsat;
void b()
{
    unsat = 14;
}
```

`main.c`

```
main()
{
    a();
}
```

If these files are compiled and linked as:

```
cc -c main.c
cc -c +z lib1.c lib2.c
ld -b lib1.o lib2.o -o liba.sl
cc main.o liba.sl -o test2
```

Using the 32-bit mode linker, `test2` executes without error. The module in `liba.sl` created from `lib2.o` is determined to be unreachable during execution, so the global symbol for `unsat` (in `lib2.o`) is not bound.

The 64-bit mode linker toolset reports an unsatisfied symbol error for `unsat` at link time or at run time if the program were made executable.

Promotion of Uninitialized Global Data Items

In the 64-bit mode linker toolset, the linker expands and promotes uninitialized global data symbols in object files linked into a shared library to global data objects, exactly like executable files. This is standard with other SVR4 systems.

The result of this change is that load-time symbol resolution for one of these objects stops at the first one encountered, instead of continuing through all loaded libraries to see if an initialized data object exists.

For example, given these source files:

`a.c`

```
int object;    /* Uninitialized global data symbol */
void a()
{
    printf ("\tobject is %d\n", object);
}
```

`b.c`

```
int object =1; /* Initialized global data symbol */
void b()
{

}
```

`main.c`

```
main()
{
a();
}
```

If these files are compiled and linked as:

```
cc -c main.c
cc -c +z a.c b.c
ld -b a.o -o libA.sl
ld -b b.o -o libB.sl
cc main.o libA.sl libB.sl -o test3
```

The 32-bit mode linker toolset produces:

```
$ test3
        object is 1
```

The 32-bit mode linker toolset defines the `object` global variable in `libA.sl` as a **storage export symbol**. The dynamic loader, when searching for a definition of `object` to satisfy the import request in `libA.sl`, does not stop with the storage export in that library. It continues to see if there is a **data export symbol** for the same symbol definition.

The 64-bit mode linker toolset produces:

```
$ test 3
        object is 0
```

The 64-bit mode linker toolset does not allow storage exports from a shared library. The uninitialized variable called `object` in `a.o` turns into a data export in `libA.sl`, with an initial value of 0. During symbol resolution for the import request from that same library, the dynamic loader stops with the first seen definition.

Symbol Searching in Dependent Libraries

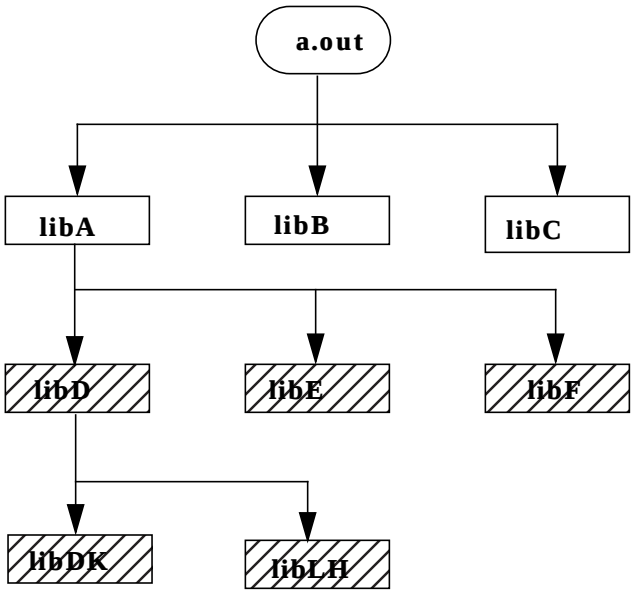
In the 64-bit mode linker toolset, the linker searches **dependent libraries** in a breadth-first order for symbol resolution. This means it searches libraries linked to the main program file before libraries linked to shared libraries. This behavior change is consistent with other SVR4 systems. The linker also searches siblings of a dependent shared library before its children. For example, using the structure described in Figure 5-25, if libD had dependent libraries libDK and libLH, libD, libE, libF, then libDK, libLH would be searched in that order. The `dlopen` library management routines and executable files (`a.out`) created by the linker with the `+std` option (default) use the **breadth-first search** order.

The 32-bit mode linker toolset searches dependent libraries in a depth-first order. This means it searches dependent shared library files in the order in which they are linked to shared libraries. The `shl_load` library management routines and executable files (`a.out`) created by the linker with the `+compat` option use the **depth-first search** order.

NOTE If you have data or function names that are duplicated in different shared libraries, the 64-bit mode linker may link in a different version of a procedure than the current release. This can lead to unexpected results.

Figure 5-25 shows an example program with shared libraries (the shaded boxes are `libA.sl` dependent libraries; and the example does not consider `libDK` or `libLH`) and compares the two search methods:

Figure 5-25 Search Order of Dependent Libraries



In 64-bit breadth-first search mode:

a.out --> libA --> libB --> libC -->libD --> libE --> libF --> LibDK --> libLH

In 32-bit depth-first search mode:

a.out --> libA --> libD --> libDK--> libLH --> libE --> LibF --> libB --> libC

The commands to build the libraries and the executable in Figure 5-25 are shown below. Note the link order of libraries in steps 2 and 3:

1. First, the dependent shared libraries for `libA` are built. (Other libraries are also built.)

```
ld -b libD.o -o libD.sl   libA dependent shared library
ld -b libE.o -o libE.sl   libA dependent shared library
ld -b libF.o -o libF.sl   libA dependent shared library
ld -b libB.o -o libB.sl
ld -b libC.o -o libC.sl
```

2. Next, `libA.o` is linked to its dependent libraries and `libA.sl` is built.

```
ld -b libA.o -lD -lE -lF -o libA.sl
```

3. Finally, `main.o` is linked to its shared libraries.

```
cc main.o -lA -lB -lC -o main
```

In the 32-bit mode linker toolset, if a procedure called `same_name()` is defined in `libD.sl` and `libB.sl`, `main` calls the procedure defined in `libD.sl`. In 64-bit linker toolset, `main` calls `same_name()` in `libB.sl`.

If you use mixed mode shared libraries, the search mechanism may produce unexpected results.

For the following command, `libA.sl` and its dependent `libB.sl` are compatibility mode libraries and `libC.sl` and `libD.sl` are standard mode libraries.

```
ld -b libF.o +compat -L.-lA -lC -o LibF.sl
```

`libF.sl` is a compatibility mode library, but is dependent `libC.sl` is a standard mode library. The linker uses depth-first searching mechanisms because the highest-level library is in compatability mode.

Mixed Mode Shared Libraries

A mixed mode shared library is a library whose children are all of one type (for example, +compat), but whose grandchildren may be of the other mode. This poses some problems when linking and loading the libraries. To help resolve this, the linker treats each library as having any mode. Therefore, when it sees a compatibility mode library, it searches for it using the 32-bit-style algorithms. For any standard mode library, it uses the 64-bit-style algorithms.Only the load order of the libraries themselves is fixed between depth-first or breadth-first.

If you use mixed mode shared libraries, you get behavior based on the first mode encountered. At runtime, the dynamic loader does a depth-first search if the dependent libraries at the highest level are compatibility mode libraries. Otherwise, it does breadth-first searching. This applies to all dependent libraries of the incomplete executable file. The loader cannot toggle back and forth between depth-first and breadth-first at the library level, so the first dependent library it sees determines which search method to use.

For example:

```
# build standard mode dlls
# libfile1.sl is a dependent of libfile2.sl


ld -b file1.o -o libfile1.sl +h libfile1.1

ld -b file2.o -o libfile2.sl +h libfile2.1 -L. -lfile1


# build compatibility mode dlls
# libfile3.sl is a dependent of libfile4.sl

ld -b file3.o -o libfile3.sl +h libfile3.1

ld -b file4.o -o libfile4.sl +h libfile4.1 -L. -lfile3 +compat

ln -s libfile1.sl libfile1.1
ln -s libfile3.sl libfile3.1


# build a dll using both standard and compatibility mode dependent
dlls
# since we didn't specify +compat, the resulting dll is a standard
mode dll


ld -b file5.o -o libfile5.sl +h libfile5.1 -L. -lfile4 -lfile2

ln -s libfile4.sl libfile4.1

ln -s libfile2.sl libfile2.1

ld main.o -L. -lfile5 -lc
```

The resulting `a.out` has standard mode dependents, `libfile5.sl` and `libc.sl`. `libfile5.sl` has two dependents,: `libfile4.sl` and `libfile2.sl`. `libfile4.sl` is a compatibility mode library, and has a dependent, `libfile3.sl`. `libfile2.sl` is a standard mode library, and has a dependent, `libfile1.sl`. The dynamic loader does a breadth-first search of all dependent libraries needed by `a.out` because the link was done without `+compat` and `libfile5.sl` is a standard

mode library. The loader uses 64-bit mode search techniques on all libraries except for `libfile3.sl`, in which case it uses 32-mode search techniques.

NOTE

Embedded path inheritance is not applied to any mixed mode shared library and its descendents. It is only applied to libraries in an `a.out` linked with `+compat`. Embedded path inheritance does not apply to a breadth-first search mechanism.

64-bit Mode Library Examples

The examples demonstrate the behavior of compatibility and standard mode shared libraries created by the 64-bit mode linker toolset:

Library Example: Creating a 64-bit Mode Compatibility Mode Shared Library.

The following example creates a compatibility mode shared library.

```
ld -b file1.o -o libfile1.sl +h libfile1.1
ld -b file2.o -o libfile2.sl +h ./libfile2.1
ld -b file3.o -o libfile3.sl +h /var/tmp/libfile3.1
ld -b file4.o -o libfile4.sl
ld -b +compat file3a.o -o libfile3a.sl -L. -lfile -lfile3 +h
libfile3a.1
ld -b +compat file2a.o -o libfile2a.sl libfile2.sl ./libfile4.sl
+b /var/tmp
elfdump -L libfile3a.sl libfile2a.sl

libfile3a.sl:

*** Dynamic Section ***

Index     Tag   Value

0  HPNeeded1:./libfile1.1 subject to dynamic path lookup
1  HPNeeded1:/var/tmp/libfile3.1  subject to dyanmic path lookup
2  Sonamelibfile3a.1
...

libfile2a.sl:

*** Dynamic Section ***

Index    Tag    Value

0  HPNeeded   0:/home/knish/./libfile2.1 not subject to dynamic path
lookup
1  HPNeeded   0:./libfile4.sl not subject to dynamic path lookup
2  Rpath      /var/tmp
...
```

Library Example: Creating a 64-bit Standard Mode Shared Library

The following example builds a standard mode library.

```
ld -b file1.o -o libfile1.sl +h libfile1.1
ld -b file2.o -o libfile2.sl +h ./libfile2.1
ld -b file3.o -o libfile3.sl +h /var/tmp/libfile3.1
ld -b file4.o -o libfile4.sl
ld -b file3a.o -o libfile3a.sl -L. -lfile1 -lfile3 +h libfile3a.1
ld -b file2a.o -o libfile2a.sl libfile2.sl ./libfile4.sl +b
/var/tmp
elfdump -L libfile3a.sl libfile2a.sl

libfile3a.sl:

*** Dynamic Section ***

Index  Tag   Value/Ptr

0   Needed   libfile1.1 subject to dynamic path lookup
1   Needed   /var/tmp/libfile3.1 not subject to dynamic path
lookup--internal pathname has a "/"
2   Soname   libfile3a.1
3   Rpath    .
...

libfile2a.sl:

*** Dynamic Section ***

Index  Tag   Value/Ptr

0   Needed ./libfile2.1 not subject to dynamic path lookup
1   Needed ./libfile4.sl not subject to dynamic path lookup
2   Rpath  /var/tmp
...
```

The dynamic loader does dynamic path searching for `libfile1.sl`.. It does not do dynamic path searching for `libfile2.sl`, `libfile3.sl`, and `libfile4.sl`.

Library example: 64-bit Mode Dynamic Path Searching

This example of dynamic path searching demonstrates differences between compatibility mode and standard mode dependent shared libraries. The example builds standard mode libraries and does a standard mode link. By default, the dynamic loader looks at the environment variables `LD_LIBRARY_PATH` and `SHLIB_PATH` to find the shared libraries.

```
# build standard mode shared libraries
#libfile1.sl is a dependent of libfile2.sl
```

```
ld -b file1.o -o libfile1.sl +h libfile1.1
ld -b file2.o -o libfile2.sl +h libfile2.1 -L. -lfile1
ld main.o -L. -lfile2 -lc

# move dependent lib so dld can't find it
# dld won't find library because we didn't set the environment
# variable LD_LIBRARY_PATH and SHLIB_PATH
# By default, dld will look at the environment variables
# LD_LIBRARY_PATH and
# SHLIB_PATH when doing dynamic path searching unless +noenvvar
# is specified

mv libfile2.sl /var/tmp
ln -s /var/tmp/libfile2.sl /var/tmp/libfile2.1
a.out
dld.sl: Unable to find library 'libfile2.1'

export SHLIB_PATH=/var/tmp
a.out
in file1
in file2
```

Library Example: 64-bit Mode Compatibility Mode Link

This example builds a compatibility mode library and does a compatibility mode link. The +s option is not specified at link time, so the dynamic loader does not look at any environment variables to do dynamic path searching.

```
# build compatibility mode dlls
# libfile1.sl is a dependent of libfile2.sl

ld -b file1.o -o libfile1.sl +h libfile1.1
ld -b file2.o -o libfile2.sl +h libfile2.1 -L. -lfile1 +compat
ln -s libfile1.sl libfile1.1
ld main.o +compat -L. -lfile2 -lc

# move dependent lib so dld can't find it. Even when we specify
SHLIB_PATH dld won't be
# able to find the dependent because we didn't link with +s

mv libfile2.sl /var/tmp
ln -s /var/tmp/libfile2.sl /var/tmp/libfile2.1
a.out
dld.sl: Unable to find library '1:./libfile2.1'
export SHLIB_PATH=/var/tmp
a.out
dld.sl: Unable to find library '1:./libfile2.1'
```

You can use `chatr +s` to enable `a.out` in `file1` and `file2`:

```
chatr +s enable a.out
```

Library Example: Using 64-bit Mode Compatibility and Standard Shared Libraries

This example mixes compatibility and standard mode shared libraries. It uses 32-it-style linking and loading for the compatibility mode libraries and 64-bit-style linking and loading for standard mode libraries.

```
# build standard mode dlls
# libfile1.sl is a dependent of libfile2

ld -b file1.o -o libfile1.sl +h libfile1.1
mkdir TMP
ld -b +b $pwd/TMP file2.o -o libfile2.sl +h libfile2.1 -L. -lfile1

# build compatibility mode dlls
# libfile3.sl is a dependent of libfile4


ld -b file3.o -o libfile3.sl +h libfile3.1
ld -b file4.o -o libfile4.sl +b $pwd/TMP +h libfile4.1 +compat -L.
-lfile3
ln -s libfile1.sl libfile1.1
ln -s libfile3.sl libfile3.1
mv libfile1.sl TMP
mv libfile3.sl TMP
cd TMP
ln -s libfile1.sl libfile1.1
ln -s libfile3.sl libfile3.1
cd ..


# link with +b so ld will use RPATH at link time to find
# libfile1.sl (standard mode dll)
# the linker will not use RPATH to find libfile3.sl
# (compatibility mode dll)
# Note that this is true in both a standard mode link and a
# compatibility mode link. The
# linker never uses RPATH to find any compatibility mode dlls


ld -b +b pwd/TMP main.o -o libfile5.sl +h libfile5.1 -L. -lfile2
-lfile4
ld: Can’t find dependent library “./libfile3.sl”
ld -b +b pwd/TMP main.o -o libfile5a.sl +h libfile5.1 -L. -lfile2
-lfile4 +compat
ld: Can’t find dependent library “./libfile3.sl”
```

Comparing Breadth-first and Depth-first Search in 64-bit Mode

For the following libraries, with the dependencies:

```
lib1.sl has dependents lib2.sl, lib3.sl, and lib4.sl
lib2.sl has dependents lib2a.sl and lib2b.sl
lib3.sl has dependents lib3a.sl and lib3b.sl
lib3a.sl has dependent lib3aa.sl
```

```
                +-->lib2a.sl
                |
      +-->lib2.sl-->lib2b.sl
      |
lib1.sl-->lib3.sl-->lib3a.sl-->lib3aa.sl
      |         |
      |         +-->lib3b.sl
      +-->lib4.sl
```

In breadth-first searching, the load order is siblings before children:

```
lib1.sl->lib2.sl->lib3.sl->lib4.sl->lib2a.sl->lib2b.sl->lib3a.sl-
>lib3b.sl->lib3aa.sl
```

In depth-first searching, the load order is children before siblings:

```
lib1.sl->lib2.sl->lib2a.sl->lib2b.sl->lib3.sl->lib3a.sl->lib3aa.s
l->lib3b.sl->lib4.sl
```

Library Example: Using RPATH with Standard Mode Shared Library

In the following example, the linker uses the embedded RPATH at link time to find the dependent library. For compatibility mode shared libraries, embedded RPATHs are ignored.

```
ld -b bar.o -o libbar.sl
ld -b foo.o -o libfoo.sl -L. -lbar +b /var/tmp
# ld should look in /var/tmp to find libbar.sl since libfoo.sl
# has an embedded RPATH of
# /var/tmp
mv libbar.sl /var/tmp
ld main.o -L. -lfoo -lc


# For compatibility mode dlls, embedded RPATHs are ignored
ld -b bar.o -o libbar.sl
ld -b foo.o -o libfoo.sl +compat -L. -lbar +b /var/tmp
# ld won't find libbar.sl since it does not look at embedded RPATHs
mv libbar.sl /var/tmp
ld main.o -L. -lfoo +compat -lc
ld: Can't find dependent library "libbar.sl"
Fatal error.
```

Linking Libraries with +b *pathlist*

The following examples compare 32-bit and 64-bit mode linking with the `ld +b` *pathlist* option. The dynamic loader uses the directory specified by the `-L` option at link time for dynamic library lookup at run time, if you do not use the `+b` option.

Library Example: Linking to Libraries with +b *path_list* in 64-bit Mode. In this example, the program `main` calls a shared library routine in `libbar.sl`. The routine in `libbar.sl` in turn calls a routine in the shared library `libme.sl`. The `+b` linker option indicates the search path for `libme.sl` when linking `libbar.sl`. (You use `+b` *path_list* with libraries specified with the `-l` *library* or `-l:`*library* options.)

```
cc -c +DD64 me.c
ld -b me.o -o libme.sl
ld -b bar.o -o libbar.sl -L. -lme +b /var/tmp
mv libme.sl /var/tmp
ld main.o -L. -lbar -lc
```

In 64-bit mode, the linker finds `libme.sl` in `/var/tmp` because the `+b /var/tmp` option is used when linking `libbar.sl`. Since `-lme` was specified when linking `libbar.sl`, `libme.sl` is subject to run-time dynamic path searching.

When linking `main.o`, the link order in the above example is:

1. `./libbar.sl` *found*
2. `./libme.sl` *not found*
3. `/var/tmp/libme.sl` *found*
4. `./libc.sl` *not found*
5. `/usr/lib/pa20_64/libc.sl` *found*

In the above example, if you type:

```
ld main.o -L. -lbar -lc
mv libme.sl /var/tmp
```

instead of:

```
mv libme.sl /var/tmp
ld main.o -L. -lbar -lc
```

the linker finds`libme.sl` in `./` at link time, and the dynamic loader finds `libme.sl` in `/var/tmp` at run time.

At run time, the dynamic loader searches paths to resolve external references made by `main` in the following order:

1. `LD_LIBRARY_PATH` to find `libbar.sl` *not found*
2. `SHLIB_PATH` to find `libbar.sl` *not found*

3. `./libbar.sl` (`./libbar.sl`) *found*
4. `LD_LIBRARY_PATH` to find `libme.sl` *not found*
5. `SHLIB_PATH` to find `libme.sl` *not found*
6. `/var/tmp/libme.sl` *found*
7. `LD_LIBRARY_PATH` to find `libc.sl` *not found*
8. `SHLIB_PATH` to find `libc.sl` *not found*
9. `./libc.sl` *not found*
10. `/usr/lib/pa20_64/libc.sl` *found*

Library Example: Linking to Libraries with +b *path_list* in 32-bit Mode. This example is the same as "Library Example: Linking to Libraries with +b path_list in 64-bit Mode", but this time the program is compiled in 32-bit mode.

```
cc -c +DD32 me.c
ld -b me.o -o libme.sl
ld -b bar.o -o libbar.sl -L. -lme +b /var/tmp
ld main.o -L. -lbar -lc
mv libme.sl /var/tmp
```

When linking `main.o`, the link order is:

1. `./libbar.sl` *found*
2. `./libme.sl` *found*
3. `./libc.sl` *not found*
4. `/usr/lib/libc.sl` *found*

In the above example, if you type:

```
mv libme.sl /var/tmp
ld main.o -L. -lbar -lc
```

instead of:

```
ld main.o -L. -lbar -lc
mv libme.sl /var/tmp
```

the linker issues the following error:

```
ld: Can't find dependent library ./libme.sl
Fatal Error
```

The linker does not look in `/var/tmp` to find shared libraries because in 32-bit mode the directories specified by `+b` *pathname* are only searched at run time.

Because `libme.sl` is specified with the `-l` option, it is subject to dynamic path searching.

At run time, the dynamic loader looks for shared libraries used by `main` in the following order:

1. `./libbar.sl` *found*
2. `/var/tmp/libme.sl` *found*
3. `./libc.sl` *not found*
4. `/usr/lib/libc.sl` *found*

NOTE In 32-bit mode, the dynamic loader does not search the default directories to find `libme.sl` at run time. In 64 bit mode, by default, the dynamic loader looks in the default directories.

6 Shared Library Management Routines

You can explicitly load and use shared libraries from your program. The linker toolset provides two families of load routines, `shl_load` and `dlopen`. The `shl_load` routines support the shared library mechanisms provided in previous version of HP-UX. The `dlopen` routines (available for 64-bit mode only) use Unix SVR4 compatible mechanism for library management.

NOTE Both families of routines support initializer and terminator routines. The 64-bit mode linker supports init/fini and 10.X release mechanisms. The 32-bit mode linker supports only the style used for the 10.X releases.

NOTE Support for `shl_load` library management routines may be discontinued in a future 64-bit HP-UX release. You are encouraged to migrate to the `dlopen` family of routines for shared library management if you use the 64-bit mode linker toolset.

Shared Library Management Routine Summaries

The following sections introduce the shared library management routines available for the HP-UX 11.00 release.

The shl_load Routine Summary

The `shl_load` family of shared library management routines are available for both the 32-bit and 64-bit mode linker.

All of the `shl_load` family routines use the same user interface in 32-bit and 64-bit mode linking.

Use the following `shl_load` routines for shared library management:

Routine	Action
`shl_load` and `cxxshl_load`	Explicitly load a shared library and a C++ shared library, respectively. They have the same syntax. `shl_load()` lets you load a compatibility or standard mode shared library. It does depth-first searching.
`shl_findsym`	Finds the address of a global symbol in a shared library.
`shl_get` and `shl_get_r`	Get information about currently loaded libraries. `shl_get_r` is a thread-safe version of `shl_get` with the same syntax.
`shl_gethandle` and `shl_gethandle_r`	Get descriptor information about a loaded shared library. `shl_gethandle_r` is a thread-safe version of `shl_gethandle` with the same syntax.

Routine	Action
`shl_definesym`	Adds a new symbol to the global shared library symbol table.
`shl_getsymbols`	Returns a list of symbols in a shared library.
`shl_unload` and `cxxshl_load`	Unload a shared library and a C++ shared library, respectively. They have the same syntax.

Except for `shl_get` and `shl_gethandle`, all these routines are thread safe.

These routines are described in the *shl_load*(3x) man page.

The dlopen Routines Summary

The `dlopen` family of shared library management routines is available only for the 64-bit linker.

The `dlopen` family of routines use Unix SVR4 shared library mechanisms.

Use the following `dl*` routines for shared library management:

Routine	Action
`dlopen`	Loads a shared library. This routine does breadth-first searching.
`dlerror`	Prints the last error message recorded by `dld`.
`dlsym`	Gets the address of a symbol in a shared library.
`dlget`	Returns information on a loaded module.
`dlmodinfo`	Returns information about a loaded module.
`dlgetname`	Retrieves the name of a loaded module given a load model descriptor.
`dlclose`	Unloads a shared library previously loaded by `dlopen()`.

All the `dlopen` routines are thread-safe.

These routines are described in the *dl**(3C) man pages.

Related Files and Commands

These commands and files provide more information about using shared library management routines.

Command/File	Action
`a.out(4)`	Executable file from assembler, compiler, and linker output.
`cc(1)`	Command to invoke the HP-UX C compiler.
`exec(2)`	System loader.
`ld(1)`	Command to invoke the linker.

Shared Library Header Files

The `shl_load` family of shared library management routines use some special data types (structures) and constants defined in the C-language header file `/usr/include/dl.h`. When using these functions from C programs, be sure to include `dl.h`:

```
#include <dl.h>
```

If you are using HP C++, also include `/opt/CC/include/CC/cxxdl.h`.

Similarly, if you are using the `dlopen` family of routines, include `/usr/include/dlfcn.h`.

```
#include <dlfnc.h>
```

If an error occurs when calling shared library management routines, the system error variable `errno` is set to an appropriate error value. Constants are defined for these error values in `/usr/include/errno.h` (see *errno*(2)). Thus, if a program checks for these error values, it must include `errno.h`:

```
#include <errno.h>
```

Throughout this section, all examples are given in C. To learn how to call these routines from C++, FORTRAN, or Pascal, refer to the inter-language calling conventions described in the compiler documentation.

Using Shared Libraries with cc and ld Options

In 32-bit mode, you can access the `shl_load` family of routines specifying the `-ldld` option on the *cc*(1) or *ld*(1) command line. In 64-bit mode, you can access the `shl_load` and `dlopen` routines by specifying either `-ldld` or `-ldl` on the command line.

Some 32-bit mode implementations do not, by default, export all symbols defined by a program (instead exporting only those symbols imported by a shared library seen at link time). Use the `-E` option to `ld` to ensure that all symbols defined in the program are available to the loaded libraries. This is the default behavior in 64-bit mode.

To create shared libraries, compile source files and link the resultant object files with the `-b` with the `cc` or `ld` command.

Initializers for Shared Libraries

A shared library can have an initialization routine—known as an **initializer**—that is called when the load module (a shared library or executable) is loaded (initializer) or explicitly unloaded (terminator). Typically, an initializer is used to initialize a shared library's data when the library is loaded.

When a program begins execution its initializers are called before any other user code is executed.This allow for setup at initialization and cleanup at termination. Also, when a shared library is explicitly loaded using `shl_load` or `dlopen` or unloaded using `shl_unload` or `dlclose`, it initializers and terminators are called at the appropriate time.

In 64-bit mode, you can specify initializers and terminators even for archive libraries or nonshared executables.

Styles of Initializers

The linker supports two different types of initializers and terminators:

- HP-UX 10.X style.
- Init/fini style.

NOTE

The 32-bit mode linker supports only the HP-UX 10.X style initializers. See “32-bit Mode Initializers” for more information.

The 64-bit mode linker supports both of these styles.See “64-bit Mode Initializers” for more information.

HP-UX-10.X-Style Initializers

HP-UX 10.X style initializers are the same type supported in all HP-UX 10.X releases. These are called both before the user’s code is started or a shared library is loaded (using `shl_load` or `dlopen`) as well as when the shared library is unloaded (using `shl_unload` or `dlclose`). The linker option `+I` is used to create this type of initializer. The function returns nothing but takes two arguments. The first is a handle to the shared library being initialized. This handle can be used in calling `shl_load` routines. The second is set to non-zero at startup and zero at program termination.

```
$ ld -b foo.o +I my_10x_init -o libfoo.sl
#include <dl.h>
void my_10x_init(shl_t handle, int loading)
{
/* handle is the shl_load API handle for the shared library being
initialized. */
/* loading is non-zero at startup and zero at termination. */
if (loading) {
... do some initializations ...
} else {
... do some clean up ...
}
}
```

NOTE Unlike 32-bit mode, the 64-bit HP-UX 10.X style initiators are called when unloading implicitly lordered shared libraries.

See “32-bit Mode Initializers” for more information on using these initiators.

Init/Fini Style Initializers

This style uses init and fini functions to handle initialization operations.

Init. Inits are called before the user's code starts or when a shared library is loaded. They are functions which take no arguments and return nothing. The C compiler pragma “init” is used to declare them. For example:

```
#pragma init “my_init”
void my_init() { ... do some initializations ... }
```

The `ld` command supports the `+init` option to specify the initializer.

Fini. Finis are called after the user's code terminates by either calling the libc exit function, returning from the main or `_start` functions, or when the shared library which contains the fini is unloaded from memory. Like inits, these also take no arguments and return nothing. The C compiler pragma “fini” is used to create them. For example:

```
#pragma fini “my_fini”
void my_fini() { ... do some clean up ... }
```

The `ld` command supports the `+fini` option to specify theterminator.

32-bit Mode Initializers

The 32-bit mode linker supports HP-UX 10.X style initializers.

This section contains the following topics:

- Using HP-UX 10.X Style Initializers
- “Declaring the Initializer with the +I Option”
- “Initializer Syntax”
- “Example: An Initializer for Each Library”
- “Example: A Common Initializer for Multiple Libraries”

Using HP-UX 10.X Style Initializers

The initializer is called for libraries that are loaded implicitly at program startup, or explicitly with `shl_load`.

When calling initializers for *implicitly* loaded libraries, the dynamic loader waits until all libraries have been loaded before calling the initializers. It calls the initializers in **depth-first order**—that is, the initializers are called in the reverse order in which the libraries are searched for symbols. All initializers are called before the main program begins execution.

When calling the initializer for *explicitly* loaded libraries, the dynamic loader waits until any dependent libraries are loaded before calling the initializers. As with implicitly loaded libraries, initializers are called in depth-first order.

Note that initializers can be disabled for explicitly loaded libraries with the `BIND_NOSTART` flag to `shl_load`. For more information, see “The shl_load and cxxshl_load Routines”.

Declaring the Initializer with the +I Option

To declare the name of the initializer, use the `+I` linker option when creating the shared library. The syntax of the `+I` option is:

+I *initializer*

where *initializer* is the initializer's name.

Multiple initializers may be called by repeating the `+I` *initializer* option.

For example, to create a shared library named `libfoo.sl` that uses an initializer named `init_foo`, use this linker command line:

```
$ ld -b -o libfoo.sl libfoo.o +I init_foo
```

Order of Execution of Multiple Initializers . Multiple initializers are executed in the same order that they appear on the command line; they are unloaded in reverse order. (This applies only to the calling order within a shared library, not across multiple shared libraries.)

NOTE

Initializers are not executed when unloading shared libraries which were implicitly loaded since the program exits without re-entering the dynamic loader to unload them. Initializers are only called during the explicit unloading of a shared library.

Initializers behave the same as other symbols; once they are bound they cannot be overridden with a new symbol through the use of `shl_definesym()` or by loading a more visible occurrence of the initializer symbol with the `BIND_FIRST` flag. What this means is that once the initializer is executed upon a load, it is guaranteed to be the same initializer that is called on an explicit unload.

Initializer Syntax

```
void initializer( shl_t handle,
                  int loading )
```

initializer The name of the initializer as specified with the `+I` linker option.

handle The *initializer* is called with this parameter set to the handle of the shared library for which it was invoked.

loading The *initializer* is called with this parameter set to –1 (true) when the shared library is loaded and 0 (false) when the library is unloaded.

The initializers cannot be defined as local definitions. Initializers cannot be hidden through the use of the `-h` option when building a shared library.

It is strongly recommended that initializers be defined with names which do not cause name collisions with other user-defined names in order to avoid overriding behavior of shared library symbol binding.

Accessing Initializers' Addresses . Prior to the HP-UX 10.0 release, initializer's addresses could be accessed through the initializer field of the shared library descriptor which is returned from a call to `shl_get()`. To support multiple initializers, the `shl_getsymbols()` routine has been enhanced to support the return of the initializer's address.

If only one initializer is specified for a given library, its address is still available through the initializer field of a shared library descriptor. If more than one initializer is specified, the initializer field will be set to `NO_INITIALIZER`. Access to multiple initializers can then be accomplished through the use of `shl_getsymbols()`. (The `shl_getsymbols()` routine can also access a single initializer.)

NOTE

`shl_getsymbols()` may not return the initializer which was invoked for a given library if a more visible initializer symbol is defined after the library being queried has been loaded. This can occur through the use of `shl_definesym()` and by explicitly loading a more visible symbol using the `BIND_FIRST` flag upon loading.

To access initializers, a new flag, `INITIALIZERS`, has been defined for the `shl_getsymbols()` routine. It can be ORed with the `NO_VALUES` and `GLOBAL_VALUES` flags. For example,

```
shl_getsymbols(handle,
               TYPE_PROCEDURE,
               INITIALIZERS | GLOBAL_VALUES,
               malloc,
               &symbol_array);
```

If the `GLOBAL_VALUES` modifier is not used and the initializer is defined in another shared library or in the program file, `shl_getsymbols()` does not find the initializer for the requested library because it is not defined within the library.

For more information on the usage of `shl_getsymbols()`, see “The shl_getsymbols Routine”.

Example: An Initializer for Each Library

One way to use initializers is to define a unique initializer for each library. For instance, the following example shows the source code for a library named `libfoo.sl` that contains an initializer named `init_foo`:

C Source for libfoo.sl

```
#include <stdio.h>
#include <dl.h>
/*
 * This is the local initializer that is called when the libfoo.sl
 * is loaded and unloaded:
 */
void init_foo(shl_t hndl, int loading)
{
  if (loading)
    printf("libfoo loaded\n");
  else
    printf("libfoo unloaded\n");
}

float in_to_cm(float in)              /* convert inches to
centimeters */
{
  return (in * 2.54);
}

float gal_to_l(float gal)            /* convert gallons to litres
*/
{
  return (gal * 3.79);
}

float oz_to_g(float oz)               /* convert ounces to grams */
{
  return (oz * 28.35);
}
```

You can use the `+I` linker option to register a routine as an initializer. Here are the commands to create `libfoo.sl` and to register `init_foo` as the initializer:

```
$ cc -Aa -c +z libfoo.c
$ ld -b -o libfoo.sl +I init_foo libfoo.o
```

To use this technique with multiple libraries, each library should have a unique initializer name. The following example program loads and unloads `libfoo.sl`.

C Source for testlib

```
#include <stdio.h>
#include <dl.h>
main()
{
float (*in_to_cm)(float), (*gal_to_l)(float), (*oz_to_g)(float);
shl_t hndl_foo;
/*
 * Load libfoo.sl and find the required symbols:
 */
if ((hndl_foo = shl_load("libfoo.sl",
     BIND_IMMEDIATE, 0)) == NULL)
   perror("shl_load: error loading libfoo.sl"), exit(1);
```

```
if (shl_findsym(&hndl_foo, “in_to_cm”, TYPE_PROCEDURE,
   (void *) &in_to_cm))
   perror(“shl_findsym: error finding in_to_cm”), exit(1);

if (shl_findsym(&hndl_foo, “gal_to_l”, TYPE_PROCEDURE,
   (void *) &gal_to_l))
   perror(“shl_findsym: error finding gal_to_l”), exit(1);

if (shl_findsym(&hndl_foo, “oz_to_g”, TYPE_PROCEDURE,
   (void *) &oz_to_g))
   perror(“shl_findsym: errror finding oz_to_g”), exit(1);
 /*
  * Call routines from libfoo.sl:
  */
 printf(“1.0in  = %5.2fcm\n”, (*in_to_cm)(1.0));
 printf(“1.0gal = %5.2fl\n”, (*gal_to_l)(1.0));
 printf(“1.0oz  = %5.2fg\n”, (*oz_to_g)(1.0));
 /*
  * Unload the library:
  */
 shl_unload(hndl_foo);
}
```

The following is the output of running the `testlib` program:

Output of testlib

```
libfoo loaded
1.0in  =  2.54cm
1.0gal =  3.79l
1.0oz  = 28.35g
libfoo unloaded
```

Example: A Common Initializer for Multiple Libraries

Rather than have a unique initializer for each library, libraries could have one initializer that calls the actual initialization code for each library. To use this technique, each library declares and references the same initializer (for example, `_INITIALIZER`), which calls the appropriate initialization code for each library.

This is easily done by defining `load` and `unload` functions in each library. When `_INITIALIZER` is called, it uses `shl_findsym` to find and call the `load` or `unload` function (depending on the value of the *loading* flag).

The following example shows the source for an `_INITIALIZER` function:

C Source for _INITIALIZER (file init.c)

```
#include <dl.h>
/*
 * Global initializer used by shared libraries that have
 * registered it:
 */
void _INITIALIZER(shl_t hand, int loading)
{
 void (*load_unload)();

 if (loading)
  shl_findsym(&hand, “load”, TYPE_PROCEDURE, (void *)
&load_unload);
 else
    shl_findsym(&hand, “unload”, TYPE_PROCEDURE, (void *)
&load_unload(;

 (*load_unload( ();           /* call the function */
}
```

The following two source files show shared libraries that have registered _INITIALIZER.

C Source for libunits.sl

```
#include <stdio.h>
#include <dl.h>
void load()                   /* called after libunits.sl loaded */
 {
  printf(“libunits.sl loaded\n”);
 }

void unload()                /* called after libunits.sl unloaded
*/
 {
  printf(“libunits.sl unloaded\n”);
 }

extern void _INITIALIZER();

float in_to_cm(float in)      /* convert inches to centimeters */
 {
  return (in * 2.54);
 }

float gal_to_l(float gal)     /* convert gallons to litres */
 {
  return (gal * 3.79);
 }

float oz_to_g(float oz)       /* convert ounces to grams */
 {
  return (oz * 28.35);
 }
```

C Source for libtwo.sl

```
#include <stdio.h>
void load()                      /* called after libtwo.sl loaded */
{
  printf("libtwo.sl loaded\n");
}
void unload()                   /* called after libtwo.sl unloaded
*/
{
  printf("libtwo.sl unloaded\n");
}

extern void _INITIALIZER();
void (*init_ptr)() = _INITIALIZER;

void foo()
{
  printf("foo called\n");
}
void bar()
{
  printf("bar called\n");
}
```

Here are the commands used to build these libraries:

```
$ cc -Aa -c +z libunits.c
$ ld -b -o libunits.sl +I _INITIALIZER libunits.o
$ cc -Aa -c +z libtwo.c
$ ld -b -o libtwo.sl +I _INITIALIZER libtwo.o
```

The following is an example program that loads these two libraries:

C Source for testlib2

```
#include <stdio.h>
#include <dl.h>
main()
{
 float (*in_to_cm)(float), (*gal_to_l)(float), (*oz_to_g)(float);
 void (*foo)(), (*bar)();
 shl_t hndl_units, hndl_two;

 /*
  * Load libunits.sl and find the required symbols:
  */
 if ((hndl_units = shl_load("libunits.sl", BIND_IMMEDIATE, 0)) ==
NULL)
   perror("shl_load: error loading libunits.sl"), exit(1);
 if (shl_findsym(&hndl_units, "in_to_cm",
     TYPE_PROCEDURE, (void *) &in_to_cm))
     perror("shl_findsym: error finding in_to_cm"), exit(1);

 if (shl_findsym(&hndl_units, "gal_to_l",
     TYPE_PROCEDURE, (void *) &gal_to_l))
     perror("shl_findsym: error finding gal_to_l"), exit(1);

 if (shl_findsym(&hndl_units, "oz_to_g",
```

```
     TYPE_PROCEDURE, (void *) &oz_to_g))
     perror("shl_findsym: errror finding oz_to_g"), exit(1);

 /*
  * Load libtwo.sl and find the required symbols:
  */
 if ((hndl_two = shl_load("libtwo.sl", BIND_IMMEDIATE, 0)) ==
NULL)
   perror("shl_load: error loading libtwo.sl"), exit(1);
 if (shl_findsym(&hndl_two, "foo", TYPE_PROCEDURE, (void *) &foo))
   perror("shl_findsym: error finding foo"), exit(1);
 if (shl_findsym(&hndl_two, "bar", TYPE_PROCEDURE, (void *) &bar))
   perror("shl_findsym: error finding bar"), exit(1);
 /*
  * Call routines from libunits.sl:
  */
 printf("1.0in  = %5.2fcm\n", (*in_to_cm)(1.0));
 printf("1.0gal = %5.2fl\n", (*gal_to_l)(1.0));
 printf("1.0oz  = %5.2fg\n", (*oz_to_g)(1.0));
 /*
  * Call routines from libtwo.sl:
  */
 (*foo)();
 (*bar)();
 /*
  * Unload the libraries so we can see messages displayed by
initializer:
  */
 shl_unload(hndl_units);
 shl_unload(hndl_two);
}
```

Here is the compiler command used to create the executable `testlib2`:

```
$ cc -Aa -Wl,-E -o testlib2 testlib2.c init.c -ldld
```

Note that the `-Wl,-E` option is required to cause the linker to export all symbols from the main program. This allows the shared libraries to find the `_INITIALIZER` function in the main executable.

Finally, the output from running `testlib2` is shown:

Output of testlib2

```
libfoo loaded
1.0in  =  2.54cm
1.0gal =  3.79l
1.0oz  = 28.35g
libfoo unloaded
```

64-bit Mode Initializers

The 64-bit mode linker support both styles of initializers:

- HP-UX 10.X style: see "HP-UX-10.X-Style Initializers" and "32-bit Mode Initializers" for more information.

- Init/Fini style: see “Init/Fini Style Initializers” and the topics described in this section:
- “Init and Fini Usage Example”
  - “Ordering Within an Executable or Shared Library”
  - “Ordering Among Executables and Shared Libraries”

Init and Fini Usage Example

This example consists of three shared libraries `lib1.sl`, `lib2.sl` and `lib3.sl`. The lib1.sl depends on `lib3.sl`. The main program (`a.out`) depends on `lib1.sl` and `lib2.sl`. Each shared library has an `init` style initializer and a `fini` style terminator. The `lib1.sl` and `lib2.sl` uses linker options (`+init` and `+fini`) to specify the initializers and terminators and `lib3.sl` uses compiler pragmas.

C source for lib1.sl (file lib1.c):

```
lib1()
{
    printf(“lib1\n”);
}

void
lib1_init()
{
    printf(“lib1_init\n”);
}

void
lib1_fini()
{
    printf(“lib1_fini\n”);
}
```

C source for lib2.sl (file lib2.c):

```
lib2()
{
    printf(“lib2\n”);
}

void
lib2_init()
{
    printf(“lib2_init\n”);
}

void
lib2_fini()
{
    printf(“lib2_fini\n”);
}
```

C source for lib3.sl (file lib3.c):

```
lib3()
{
    printf("lib3\n");
}

#pragma init "lib3_init"

void
lib3_init()
{
    printf("lib3_init\n");
}

#pragma fini "lib3_fini"

void
lib3_fini()
{
    printf("lib3_fini\n");

}
```

Commands used to build these libraries:

```
$ cc +DD64 lib1.c lib2.c lib3.c main.c -c;
$ ld -b lib3.o -o lib3.sl;
$ ld -b +init lib2_init +fini lib2_fini lib2.o -o lib2.sl;
$ ld -b +init lib1_init +fini lib1_fini lib1.o ./lib3.sl -o \
lib1.sl;
$ cc -L. +DD64 main.o -l1 -l2 -lc;
```

Output from running a.out:

```
lib2_init
lib3_init
lib1_init
lib1
lib2
lib3
lib1_fini
lib3_fini
lib2_fini
```

Ordering Within an Executable or Shared Library

Multiple initializers/terminators within the same load module (an executable or shared library) are called in an order following these rules:

- Inits in `.o` (object) files or `.a` (archive) files are called in the reverse order of the link line.
- Finis in `.o` or `.a` files are called in forward order of the link line.

- HP-UX 10.X style initializers are called in forward order of the +I options specified on the link line when loading a shared library. They are then called in reverse order when unloading the library.
- HP-UX 10.X style initializers are called after inits and before finis.
- Any inits or finis in archive (.a) files are called only if the .o which contains it is used during the link. Use the linker -v option to determine which .o files within an archive file were used.
- Shared libraries on the link line (dependent libraries) follow the ordering described in "Ordering Among Executables and Shared Libraries".

For example, the linker command:

```
$ ld -b first_64bit.o -l:libfoo.sl second_64bit.o my_64bit.a +I
first_10x_init +I second_10x_init -o libbar.sl
```

results in the following order when library is loaded:

1. inits from any .o files used in my_64bit.a
2. inits in second_64bit.o
3. inits in first_64bit.o
4. first_10x_init
5. second_10x_init

and the following order when library is unloaded:

1. second_10x_init
2. first_10x_init
3. finis in first_64bit.o
4. finis in second_64bit.o
5. finis from any .o files used in my_64bit.a

NOTE libfoo.sl is ignored in this example. It follows the rules in "Ordering Among Executables and Shared Libraries".

Ordering Among Executables and Shared Libraries

When multiple load modules have initializers/terminators, the following rules apply to ordering:

- When loading, the inits and HP-UX 10.X initializers of any dependent libraries are called before the ones in the current library.
- When unloading, the finis and HP-UX 10.X initializers of any dependent libraries are called after the finis of the current library.
- If a shared library is itself a dependent of one of its dependents (a “circular” dependency), no ordering between them is guaranteed.

For example, given three libraries: `libA.sl`, `libB.sl`, `libC.sl`. If `libA.sl` were linked as (`libB.sl` and `libC.sl` are “dependent” libraries of `libA.sl`):

```
$ ld -b foo.o -lB -lC -o libA.sl
```

One possible ordering while loading is:

- inits in C
- inits in B
- inits in A

and while unloading is:

- finis inA
- finis in B
- finis in C

The shl_load Shared Library Management Routines

This section describes the `shl_load` family of shared library management routines.

NOTE

You can use these routines in both 32-bit and 64-bit mode. Support for these routines may be discontinues in a future 64-bit HP-UX release. If you use these routines in 64-bit mode, consider converting your programs to the `dl*` family of shared library management routines.

The shl_load and cxxshl_load Routines

Explicitly loads a library.

Syntax

```
shl_t shl_load( const char * path,
                int flags,
                long address )
```

Parameters

path A null-terminated character string containing the path name of the shared library to load.

flags Specifies when the symbols in the library should be bound to addresses. It must be one of these values, defined in `<dl.h>`:

`BIND_IMMEDIATE`

Bind the addresses of all symbols immediately upon loading the library.

`BIND_DEFERRED`

Bind the addresses when they are first referenced.

Be aware that `BIND_IMMEDIATE` causes the binding of all symbols, and the resolution of all imports, even from older versioned modules in the shared library. If symbols are not accessible because they come from old modules, they are unresolved and `shl_load` may fail.

In addition to the above values, the *flags* parameter can be ORed with the following values:

`BIND_NONFATAL`

Allow binding of unresolved symbols.

`BIND_VERBOSE`

Make dynamic loader display verbose messages when binding symbols.

`BIND_FIRST`

Insert the loaded library before all others in the current link order.

`DYNAMIC_PATH`

Causes the dynamic loader to perform dynamic library searching when loading the library. The `+s` and `+b` options to the `ld` command determine the directories the linker searches. This is the default mode for the 64-bit mode linker if `+compat` linker option is not specified.

`BIND_NOSTART`

Causes the dynamic loader to *not* call the initializer, even if one is declared for the library, when the library is loaded or on a future call to `shl_load` or `dlopen`. This also inhibits a call to the initializer when the library is unloaded.

`BIND_RESTRICTED`

Causes the search for a symbol definition to be restricted to those symbols that were visible when the library was loaded.

`BIND_TOGETHER`

Causes the library being loaded and all its dependent libraries to be bound together rather than each independently. Use this when you have interdependent libraries and you are using `BIND_FIRST`.

`BIND_BREADTH_FIRST`

64-bit mode only:

Causes the dependent libraries to be loaded breadth first. By default, the 64-bit mode `shl_load` loads dependent libraries depth-first.

These *flags* are discussed in detail in“shl_load Example”.

address Specifies the virtual address at which to attach the library. Set this parameter to `0` (zero) to tell the system to choose the best location. This argument is currently ignored; mapping a library at a user-defined address is not currently supported.

Return Value

If successful, `shl_load` returns a **shared library handle** of type `shl_t`. This address can be used in subsequent calls to `shl_close`, `shl_findsym`, `shl_gethandle`, and `shl_gethandle_r`. Otherwise, `shl_load` returns a shared library handle of `NULL` and sets `errno` to one of these error codes (from `<errno.h>`):

`ENOEXEC` The specified *path* is not a shared library, or a format error was detected in this or another library.

`ENOSYM` A symbol needed by this library or another library which this library depends on could not be found.

`ENOMEM` There is insufficient room in the address space to load the shared library.

`EINVAL` The requested shared library address was invalid.

`ENOENT` The specified *path* does not exist.

`EACCESS` Read or execute permission is denied for the specified *path*.

Description

A program needs to explicitly load a library only if the library was not linked with the program. This typically occurs only when the library cannot be known at link time — for example, when writing programs that must support future graphics devices.

However, programs are not restricted to using shared libraries only in that situation. For example, rather than linking with any required libraries, a program could explicitly load libraries as they are needed. One possible reason for doing this is to minimize virtual memory overhead. To keep virtual memory resource usage to a minimum, a program could load libraries with `shl_load` and unload with `shl_unload` when the library is no longer needed. However, it is normally not necessary to incur the programming overhead of loading and unloading libraries yourself for the sole reason of managing system resources.

Note that if shared library initializers have been declared for an explicitly loaded library, they are called after the library is loaded. For details, see “Initializers for Shared Libraries”.

To explicitly load a shared library, use the `shl_load` routine. If loading a C++ library, use the `cxxshl_load` routine. This ensures that constructors of nonlocal static objects are executed when the library is loaded. The syntax of `cxxshl_load` is the same as that of `shl_load`.

In 64-bit mode, `shl_load` lets you load a compatibility or standard mode shared libraries. The `BIND_BREADTH_FIRST` flag overrides the default depth-first loading mechanism.

shl_load Usage

Since the library was not specified at link time, the program must get the library name at run time. Here are some practical ways to do this:

- Hard-code the library name into the program (the easiest method).
- Get the library name from an environment variable using the `getenv` library routine (see *getenv*(3C)).
- Get the library path name from the command line through `argv`.
- Read the library name from a configuration file.
- Prompt for the library path name at run time.

If successful, `shl_load` returns a shared library handle (of type `shl_t`), which uniquely identifies the library. This handle can then be passed to the `shl_findsym` or `shl_unload` routine.

Once a library is explicitly loaded, use the `shl_findsym` routine to get pointers to functions or data contained in the library; then call or reference them through the pointers. This is described in detail in“The shl_findsym Routine”.

shl_load Example

The following example shows the source for a function named `load_lib` that explicitly loads a library specified by the user. The user can specify the library in the environment variable SHLPATH or as the only argument on the command line. If the user chooses neither of these methods, the function prompts for the library path name.

The function then attempts to load the specified library. If successful, it returns the shared library handle, of type `shl_t`. If an error occurs, it displays an error message and exits. This function is used later in “The shl_findsym Routine”.

load_lib — Function to Load a Shared Library

```
#include        <stdio.h>      /* contains standard I/O
defs           */
#include        <stdlib.h>     /* contains getenv
definition          */
#include      <dl.h>       /* contains shared library type defs
*/

shl_t load_lib(int argc,
               char * argv[])   /* pass argc and argv from main */
{
  shl_t   lib_handle;             /* temporarily holds library
handle  */
  char    lib_path[MAXPATHLEN];   /* holds library path
name           */
  char    *env_ptr;               /* points to SHLPATH variable
value  */
  /*
   * Get the shared library path name:
   */
  if (argc > 1)              /* library path given on command line
*/
    strcpy(lib_path, argv[1]);
  else                       /* get lib_path from SHLPATH variable
*/
    {
      env_ptr = getenv(“SHLPATH”);
      if (env_ptr != NULL)
        strcpy(lib_path, env_ptr);
     else                    /* prompt user for shared library path
*/
        {
          printf(“Shared library to use >> “);
          scanf(“%s”, lib_path);
        }
```

```
    }
  /*
   * Dynamically load the shared library using BIND_IMMEDIATE
binding:
   */
  lib_handle = shl_load( lib_path, BIND_IMMEDIATE, 0);
  if (lib_handle == NULL)
    perror("shl_load: error loading library"), exit(1);
  return lib_handle;
}
```

BIND_NONFATAL Modifier

If you load a shared library with the BIND_IMMEDIATE flag and the library contains unresolved symbols, the load fails and sets errno to ENOSYM. ORing BIND_NONFATAL with BIND_IMMEDIATE causes shl_load to allow the binding of unresolved symbols to be deferred if their later use can be detected — for example:

```
shl_t libH;
   . . .
libH = shl_load("libxyz.sl", BIND_IMMEDIATE | BIND_NONFATAL, 0);
```

However, data symbol binding cannot be deferred, so using the BIND_NONFATAL modifier does not allow the binding of unresolved data symbols.

BIND_VERBOSE Modifier

If BIND_VERBOSE is ORed with the *flags* parameter, the dynamic loader displays messages for all unresolved symbols. This option is useful to see exactly which symbols cannot be bound. Typically, you would use this with BIND_IMMEDIATE to debug unresolved symbols — for example:

```
shl_t libH;
   . . .
libH = shl_load("libxyz.sl", BIND_IMMEDIATE | BIND_VERBOSE, 0);
```

BIND_FIRST Modifier

If BIND_FIRST is ORed with the *flags* parameter, the loaded library is inserted before all other loaded shared libraries in the symbol resolution search order. This has the same effect as placing the library first in the link order — that is, the library is searched before other libraries when resolving symbols. This is used with either BIND_IMMEDIATE or BIND_DEFERRED — for example:

```
shl_t libH;
  . . .
libH = shl_load("libpdq.sl", BIND_DEFERRED | BIND_FIRST, 0);
```

BIND_FIRST is typically used when you want to make the symbols in a particular library more visible than the symbols of the same name in other libraries. Compare this with the default behavior, which is to *append* loaded libraries to the link order.

DYNAMIC_PATH Modifier

The flag DYNAMIC_PATH can also be ORed with the *flags* parameter, causing the dynamic loader to search for the library using a path list specified by the +b option at link time or the SHLIB_PATH environment variable at run time.

BIND_NOSTART Modifier

The flag BIND_NOSTART inhibits execution of initializers for the library.

BIND_RESTRICTED Modifier

This flag is most useful with the BIND_DEFERRED flag; it has no effect with BIND_IMMEDIATE. It is also useful with the BIND_NONFATAL flag.

When used with only the BIND_DEFERRED flag, it has this behavior: When a symbol is referenced and needs to be bound, this flag causes the search for the symbol definition to be restricted to those symbols that were visible when the library was loaded. If a symbol definition cannot be found within this restricted set, it results in a run-time symbol-binding error.

When used with BIND_DEFERRED and the BIND_NONFATAL modifier, it has the same behavior, except that when a symbol definition cannot be found, the dynamic loader will then look in the global symbol set. If a definition still cannot be found within the global set, a run-time symbol-binding error occurs.

BIND_TOGETHER Modifier

BIND_TOGETHER modifies the behavior of BIND_FIRST. When the library being loaded has dependencies, BIND_FIRST causes each dependent library to be loaded and bound separately. If the libraries have interdependencies, the load may fail because the needed symbols are not available when needed.

BIND_FIRST | BIND_TOGETHER causes the library being loaded and its dependent libraries to be bound all at the same time, thereby resolving interdependencies. If you are not using BIND_FIRST, libraries are bound together by default so this option has no effect.

BIND_BREADTH_FIRST Modifier

64-bit mode only:

This flag causes the dependent libraries to be loaded breadth first. By default, the 64-bit mode shl_load loads dependent libraries depth-first. This modifier overrides the default load order.

Binding Flags Examples

Suppose you have the libraries libE.sl, libF.sl, and libG.sl. The libE library depends on libF and libF depends on libG. In addition, libG depends on libF—libF and libG are interdependent. Your program loads libE.sl with shl_load().

When using BIND_DEFERRED or BIND_IMMEDIATE *without* BIND_FIRST, these libraries are loaded such that all symbols are visible and the interdependencies are resolved:

```
shl_t libE;
libE = shl_load("libE.sl", BIND_IMMEDIATE, 0);
```
shl_load succeeds.

When using BIND_IMMEDIATE | BIND_FIRST, however, libG is loaded and bound first and since it depends on libF, an error results because the needed symbols in libF are not yet available:

```
libE = shl_load("libE.sl", BIND_IMMEDIATE | BIND_FIRST, 0);
```
shl_load fails.

Using BIND_IMMEDIATE | BIND_FIRST | BIND_TOGETHER loads libE, libF, and libG together and correctly resolves all symbols:

```
libE = shl_load("libE.sl", BIND_IMMEDIATE | BIND_FIRST | BIND_TOG
ETHER, 0);
```
shl_load succeeds.

The shl_findsym Routine

Obtains the address of an exported symbol from a shared library. To call a routine or access data in an explicitly loaded library, first get the address of the routine or data with shl_findsym.

Syntax

```
int shl_findsym( shl_t * handle,
                 const char * sym,
                 short type,
                 void * value )
```

Parameters

handle A *pointer* to a shared library handle of the library to search for the symbol name *sym*. This handle could be obtained from the `shl_get` routine (described in the "The shl_get and shl_get_r Routines"). *handle* can also point to:

`NULL` If a pointer to `NULL` is specified, `shl_findsym` searches *all* loaded libraries for *sym*. If *sym* is found, `shl_findsym` sets *handle* to a pointer to the handle of the shared library containing *sym*. This is useful for determining which library a symbol resides in. For example, the following code sets `handle` to a pointer to the handle of the library containing symbol `_foo`:

```
shl_t handle;
handle = NULL;
shl_findsym(&handle,"_foo",...);
```

`PROG_HANDLE` This constant, defined in `dl.h`, tells `shl_findsym` to search for the symbol in the program itself. This way, any symbols exported from the program can be accessed explicitly.

sym A null-terminated character string containing the name of the symbol to search for.

type The type of symbol to look for. It must be one of the following values, defined in `<dl.h>`:

`TYPE_PROCEDURE`

Look for a function or procedure.

`TYPE_DATA`

	Look for a symbol in the data segment (for example, variables).
	`TYPE_UNDEFINED`
	Look for *any* symbol.
	`TYPE_STORAGE`
	32-bit mode only.
	`TYPE_TSTORAGE`
	32-bit mode only.
value	A pointer in which `shl_findsym` stores the address of *sym*, if found.

Return Value

If successful, `shl_findsym` returns an integer (`int`) value zero. If `shl_findsym` cannot find *sym*, it returns –1 and sets `errno` to zero. If any other errors occur, `shl_findsym` returns –1 and sets `errno` to one of these values (defined in `<errno.h>`):

`ENOEXEC`	A format error was detected in the specified library.
`ENOSYM`	A symbol on which *sym* depends could not be found.
`EINVAL`	The specified *handle* is invalid.

Description

To call a routine or access data in an explicitly loaded library, first get the address of the routine or data with `shl_findsym`.

To call a routine in an explicitly loaded library

1. declare a pointer to a function of the same type as the function in the shared library
2. using `shl_findsym` with the *type* parameter set to `TYPE_PROCEDURE`, find the symbol in the shared library and assign its address to the function pointer declared in Step 1
3. call the pointer to the function obtained in Step 2, with the correct number and type of arguments

To access data in an explicitly loaded library

1. declare a pointer to a data structure of the same type as the data structure to access in the library
2. using `shl_findsym` with the *type* parameter set to `TYPE_DATA`, find the symbol in the shared library and assign its address to the pointer declared in Step 1
3. access the data through the pointer obtained in Step 2

shl_findsym Example

Suppose you have a set of libraries that output to various graphics devices. Each graphics device has its own library. Although the actual code in each library varies, the routines in these shared libraries have the same name and parameters, and the global data is the same. For instance, they all have these routines and data:

`gopen()`	opens the graphics device for output
`gclose()`	closes the graphics device
`move2d(`*x*`,`*y*`)`	moves to pixel location *x*,*y*
`draw2d(`*x*`,`*y*`)`	draws to pixel location *x*,*y* from current *x*,*y*
`maxX`	contains the maximum X pixel location on the output device
`maxY`	contains the maximum Y pixel location on the output device

The following example shows a C program that can load any supported graphics library at run time, and call the routines and access data in the library. The program calls `load_lib` (see “load_lib — Function to Load a Shared Library”) to load the library.

Load a Shared Library and Call Its Routines and Access Its Data

```
#include <stdio.h>      /* contains standard I/O defs        */
#include <stdlib.h>     /* contains getenv definition        */
#include <dl.h>         /* contains shared library type defs */
/*
 * Define linker symbols:
 */

#define GOPEN   “gopen”
#define GCLOSE  “gclose”
#define MOVE2D  “move2d”
#define DRAW2D  “draw2d”
#define MAXX    “maxX”
#define MAXY    “maxY”
```

```
shl_t   load_lib(int argc, char * argv[]);
main(int argc,
     char * argv[])
{
  shl_t lib_handle;           /* handle of shared library       */
  int   (*gopen)(void);       /* opens the graphics device      */
  int   (*gclose)(void);      /* closes the graphics device     */
  int   (*move2d)(int, int);  /* moves to specified x,y location */
  int   (*draw2d)(int, int);  /* draw line to specified x,y location*/
  int   *maxX;                /* maximum X pixel on device      */
  int   *maxY;                /* maximum Y pixel on device      */

  lib_handle = load_lib(argc, argv);  /* load required shared library */
  /*
   * Get addresses of all functions and data that will be used:
   */
  if (shl_findsym(&lib_handle, GOPEN, TYPE_PROCEDURE, (void *) &gopen))
    perror("shl_findsym: error finding function gopen"), exit(1);
  if (shl_findsym(&lib_handle, GCLOSE, TYPE_PROCEDURE, (void *) &gclose))
    perror("shl_findsym: error finding function gclose"), exit(1);
  if (shl_findsym(&lib_handle, MOVE2D, TYPE_PROCEDURE, (void *) &move2d))
    perror("shl_findsym: error finding function move2d"), exit(1);
  if (shl_findsym(&lib_handle, DRAW2D, TYPE_PROCEDURE, (void *) &draw2d))
    perror("shl_findsym: error finding function draw2d"), exit(1);
  if (shl_findsym(&lib_handle, MAXX, TYPE_DATA, (void *) &maxX))
    perror("shl_findsym: error finding data maxX"), exit(1);
  if (shl_findsym(&lib_handle, MAXY, TYPE_DATA, (void *) &maxY))
    perror("shl_findsym: error finding data maxY"), exit(1);
  /*
   * Using the routines, draw a line from (0,0) to (maxX,maxY):
   */
  (*gopen)();                 /* open the graphics device       */
  (*move2d)(0,0);             /* move to pixel 0,0              */
  (*draw2d)(*maxX,*maxY);     /* draw line to maxX,maxY pixel   */
  (*gclose)();                /* close the graphics device      */
}
```

Shown below is the compile line for this program, along with the commands to set SHLPATH appropriately before running the program. SHLPATH is declared and used by `load_lib()`, defined in"The shl_load and cxxshl_load Routines" example. Notice that `load_lib()` is compiled here along with this program. Finally, this example assumes you have created a graphics library, `libgrphdd.sl`:

```
$ cc -Aa -o drawline shl_findsym.c load_lib.c -ldld
$ SHLPATH=/usr/lib/libgrphdd.sl
$ export SHLPATH
$ drawline
```

The shl_get and shl_get_r Routines

Obtains information on the currently loaded libraries.

Syntax

```
int shl_get( int index,
             struct shl_descriptor **desc )
```

Parameters

index

Specifies an ordinal number of the shared library in the process. For libraries loaded *implicitly* (at startup time), *index* is the ordinal number of the library as it appeared on the command line. For example, if `libc` was the first library specified on the `ld` command line, then `libc` has an *index* of 1. For explicitly loaded libraries, *index* corresponds to the order in which the libraries were loaded, starting after the ordinal number of the last implicitly loaded library. Two *index* values have special meaning:

0	Refers to the main program itself
–1	Refers to the dynamic loader (`dld.sl`).

A shared library's *index* can be modified during program execution by either of the following events:

- The program loads a shared library with the `BIND_FIRST` modifier to `shl_load`. This increments all the shared library indexes by one.
- The program unloads a shared library with `shl_unload`. Any libraries following the unloaded library have their index decremented by one.

desc

Returns a pointer to a statically allocated buffer (`struct shl_descriptor **`) containing a shared library descriptor. The structure contains these important fields:

`tstart`	The start address (`unsigned long`) of the shared library text segment.
`tend`	The end address (`unsigned long`) of the shared library text segment.
`dstart`	The start address (`unsigned long`) of the shared library data segment.

`dend`	The end address (`unsigned long`) of the shared library bss segment. The data and bss segments together form a contiguous memory block starting at `dstart` and ending at `dend`.
`handle`	The shared library's handle (type `shl_t`).
`filename`	A character array containing the library's path name as specified at link time or at explicit load time.
`initializer`	A pointer to the shared library's initializer routine (see “Initializers for Shared Libraries”. It is `NULL` if there is no initializer. This field is useful for calling the initializer if it was disabled by the `BIND_NOSTART` flag to `shl_load`. If the shared library has multiple initializers, this field will also be set to `NULL`. Multiple initializers can be found with `shl_getsymbols`, described later in this chapter.

This buffer is statically allocated. Therefore, if a program intends to use any of the members of the structure, the program should make a copy of the structure before the next call to `shl_get`. Otherwise, `shl_get` will overwrite the static buffer when called again.

Return Value

If successful, `shl_get` returns an integer value 0. If the *index* value exceeds the number of currently loaded libraries, `shl_get` returns –1 and sets `errno` to `EINVAL`.

Description

To obtain information on currently loaded libraries, use the `shl_get` function. If you are programming in a threaded environment, use the thread-safe version `shl_get_r` which is the same as `shl_get` in all other respects. (See *Programming with Threads on HP-UX* for more information about threads.)

Other than obtaining interesting information, this routine is of little use to most programmers. A typical use might be to display the names and starting/ending address of all shared libraries in a process's virtual memory address space.

Example

The function `show_loaded_libs` shown below displays the name and start and end address of the text and data/bss segments the library occupies in a process's virtual address space.

show_loaded_libs — Display Library Information

```
#include        <stdio.h>      /* contains standard I/O defs          */
#include        <dl.h>         /* contains shared library type defs   */
void    show_loaded_libs(void)
{
int     idx;
struct  shl_descriptor  *desc;

  printf("SUMMARY of currently loaded libraries:\n");
  printf("%-25s  %10s  %10s  %10s  %10s\n",
      "___library___", "_tstart_", "__tend__", "_dstart_", "__dend__");

  idx = 0;
  for (idx = 0; shl_get(idx, &desc) != -1; idx++)
    printf("%-25s  %#10lx  %#10lx  %#10lx  %#10lx\n",
     desc->filename, desc->tstart, desc->tend, desc->dstart, desc->dend);
}
```

Calling this function from a C program compiled with shared `libc` and `libdld` produced the following output:

```
SUMMARY of currently loaded libraries:
___library___          _tstart_    __tend__    _dstart_    __dend__
./a.out                  0x1000      0x1918  0x40000000  0x40000200
/usr/lib/libdld.sl  0x800ac800  0x800ad000  0x6df62800  0x6df63000
/usr/lib/libc.sl    0x80003800  0x80091000  0x6df63000  0x6df85000
```

The shl_gethandle and shl_gethandle_r Routines

Returns descriptor information about a loaded shared library.

Syntax

```
int shl_gethandle( shl_t handle,
                   struct shl_descriptor **desc )
```

Parameters

handle The handle of the shared library you want information about. This *handle* is the same as that returned by `shl_load`.

desc Points to shared library descriptor information — the same information returned by the `shl_get` routine. The buffer used to store this *desc* information is static, meaning that subsequent calls to `shl_gethandle` will overwrite the same area with new data. Therefore, if you need to save the *desc* information, copy it elsewhere before calling `shl_gethandle` again.

Return Value

If *handle* is not valid, the routine returns –1 and sets `errno` to `EINVAL`. Otherwise, `shl_gethandle` returns 0.

Description

The `shl_gethandle` routine returns descriptor information about a loaded shared library. If you are programming in a threaded environment, use the thread-safe version `shl_gethandle_r` which is the same as `shl_gethandle` in all other respects. (See *Programming with Threads on HP-UX* for more information about threads.)

Example

The following function named `show_lib_info` displays information about a shared library, given the library's handle.

show_lib_info — Display Information for a Shared Library

```
#include <stdio.h>
#include <dl.h>
```

```
int show_lib_info(shl_t libH)
{
struct shl_descriptor *desc;

  if (shl_gethandle(libH, &desc) == -1)
  {
    fprintf(stderr, "Invalid library handle.\\n");
    return -1;
  }
  printf("library path:    %s\\n", desc->filename);
  printf("text start:      %#10lx\\n", desc->tstart);
  printf("text end:        %#10lx\\n", desc->tend);
  printf("data start:      %#10lx\\n", desc->dstart);
  printf("data end:        %#10lx\\n", desc->dend);
  return 0;
}
```

The shl_definesym Routine

Adds new symbols to the global shared library symbol table.

Syntax

```
int shl_definesym( const char *sym,
                   short type,
                   long value,
                   int flags )
```

Parameters

sym — A null-terminated string containing the name of the symbol to change or to add to the process's shared library symbol table.

type — The type of symbol — either TYPE_PROCEDURE or TYPE_DATA.

value — If *value* falls in the address range of a currently loaded library, an association will be made and the symbol is undefined when the library is unloaded. (Note that memory dynamically allocated with *malloc*(3C) does *not* fall in the range of *any* library.) The defined symbol may be overridden by a subsequent call to this routine or by loading a more visible library that provides a definition for the symbol.

flags — Must be set to zero.

Return Value

If successful, `shl_definesym` returns 0. Otherwise, it returns –1 and sets `errno` accordingly. See *shl_definesym*(3X) for details.

Description

The `shl_definesym` function allows you to add a new symbol to the global shared library symbol table. Use of this routine will be unnecessary for most programmers.

There are two main reasons to add or change shared library symbol table entries:

- to generate symbol definitions as the program runs — for example, aliasing one symbol with another
- to override a current definition

Symbol definitions in the incomplete executable may also be redefined with certain restrictions:

- The incomplete executable always uses its own definition for any data (storage) symbol, even if a more visible one is provided.
- The incomplete executable only uses a more visible code symbol if the main program itself does not provide a definition.

The shl_getsymbols Routine

The `shl_getsymbols` function retrieves symbols that are imported (referenced) or exported (defined) by a shared library. This information is returned in an allocated array of records, one for each symbol. Most programmers do not need to use this routine.

Syntax

```
int shl_getsymbols( shl_t handle,
                    short type,
                    int flags,
                    void * (*memfunc)(),
                    struct shl_symbol **symbols )
```

Parameters

handle The handle of the shared library whose symbols you want to retrieve. If *handle* is `NULL`, `shl_getsymbols` returns symbols that were defined with the `shl_definesym` routine.

type Defines the type of symbol to retrieve. It must be one of the following values, which are defined as constants in `<dl.h>`:

`TYPE_PROCEDURE`

Retrieve only function or procedure symbols.

`TYPE_DATA`

Retrieve only symbols from the data segment (for example, variables).

`TYPE_UNDEFINED`

Retrieve *all* symbols, regardless of type.

`TYPE_STORAGE`

32-bit mode only.

`TYPE_TSTORAGE`

32-bit mode only.

flags Defines whether to retrieve import or export symbols from the library. An **import symbol** is an external reference made from a library. An **export symbol** is a symbol definition that is referenced outside the library. In addition, any symbol defined by `shl_definesym` is an export symbol. Set this argument to one of the following values (defined in `<dl.h>`):

`IMPORT_SYMBOLS`

To return import symbols.

`EXPORT_SYMBOLS`

To return export symbols.

`INITIALIZERS`

To return initializer symbols.

One of the following modifiers can be ORed with both the EXPORT_SYMBOLS and the INITIALIZERS flags:

NO_VALUES — Do not calculate the value field of the shl_symbol structure for symbols. The value field has an undefined value.

GLOBAL_VALUES — For symbols that are defined in multiple libraries, this flag causes shl_getsymbols to return the most-visible occurrence, and to set the value and handle fields of the shl_symbol structure (defined in the description of the *symbols* parameter).

memfunc — Points to a function that has the same interface (calling conventions and return value) as *malloc*(3C). The shl_getsymbols function uses this function to allocate memory to store the array of symbol records, *symbols*.

symbols — This points to an array of symbol records for all symbols that match the criteria determined by the *type* and *value* parameters. The type of these records is struct shl_symbol, defined in <dl.h> as:

```
struct shl_symbol {
  char * name;
  short  type;
  void * value;
  shl_t  handle;
};
```

The members of this structure are described in "The shl_symbol Structure".

Return Value

If successful, shl_getsymbols returns the number of symbols found; otherwise, –1 is returned and shl_getsymbols sets errno to one of these values:

ENOEXEC — A format error was detected in the specified library.

ENOSYM	Some symbol required by the shared library could not be found.
EINVAL	The specified *handle* is invalid.
ENOMEM	*memfunc* failed to allocate the requested memory.

The shl_symbol Structure

The members of the shl_symbol structure are defined as follows:

name	Contains the name of a symbol.
type	Contains the symbol's type: TYPE_PROCEDURE, TYPE_DATA, or TYPE_STORAGE. TYPE_STORAGE is a data symbol used for C uninitialized global variables or FORTRAN common blocks.
value	Contains the symbol's address. It is valid only if EXPORT_SYMBOLS is specified without the NO_VALUES modifier.
handle	Contains the handle of the shared library in which the symbol is found, or NULL in the case of symbols defined by shl_definesym. It is valid only if EXPORT_SYMBOLS or INITIALIZERS were requested without the NO_VALUES modifier. It is especially useful when used with the GLOBAL_VALUES modifier, allowing you to determine the library in which the most-visible definition of a symbol occurs.

shl_getsymbols Example

"show_symbols — Display Shared Library Symbols" shows the source for a function named show_symbols that displays shared library symbols. The syntax of this routine is defined as:

```
int show_symbols(shl_t  hndl,
                 short  type,
                 int    flags)
```

hndl	The handle of the shared library whose symbols you want to display.

type The type of symbol you want to display. This is the same as the *type* parameter to shl_getsymbols and can have these values: TYPE_PROCEDURE, TYPE_DATA, or TYPE_UNDEFINED. If it is TYPE_UNDEFINED, show_symbols displays the type of each symbol.

flags This is the same as the *flags* parameter. It can have the value EXPORT_SYMBOLS or IMPORT_SYMBOLS. In addition, it can be ORed with NO_VALUES or GLOBAL_VALUES. If EXPORT_SYMBOLS is specified without being ORed with NO_VALUES, show_symbols displays the address of each symbol.

show_symbols — Display Shared Library Symbols

```
#include <dl.h>
#include <stdio.h>
#include <stdlib.h>
int show_symbols(shl_t   hndl,
                 short   type,
                 int     flags)
{
  int   num_symbols, sym_idx;
  struct shl_symbol *symbols, *orig_symbols;

  num_symbols = shl_getsymbols(hndl, type, flags, malloc,
&symbols);
  if (num_symbols < 0) {
    printf("shl_getsymbols failed\n");
    exit(1);
  }
  orig_symbols = symbols;
  for (sym_idx = 0; sym_idx < num_symbols; sym_idx++)
  {
   printf("    %-30s", symbols->name); /* display symbol name    */
   if (type == TYPE_UNDEFINED) /* display type if TYPE_UNDEFINED */
      switch (symbols->type) {
        case TYPE_PROCEDURE:
          printf("  PROCEDURE");
          break;
        case TYPE_DATA:
          printf("  DATA     ");
          break;
        case TYPE_STORAGE:
          printf("  STORAGE  ");
        }
   if ((flags & EXPORT_SYMBOLS)      /* export symbols requested
*/
        && (flags & NO_VALUES)==0)      /* NO_VALUES was NOT
specified */
     printf("  0x%8X", symbols->value); /* so display symbol's
address */
   printf("\n");                    /* terminate output line
*/
   symbols++;                      /* move to next symbol record
```

```
*/
    }
 free(orig_symbols);                /* free memory allocated by
malloc */
 return num_symbols;               /* return the number of symbols
*/
}
```

The following example shows the source for a program named show_all.c that calls show_symbols to show all imported and exported symbols for every loaded shared library. It uses shl_get to get the library handles of all loaded libraries.

show_all — Use show_symbols to Show All Symbols

```
#include <dl.h>
#include <stdio.h>
/* prototype for show_syms */
int show_syms(shl_t hndl, short type, int flags);
main()
{
 int   idx, num_syms;
 struct shl_descriptor * desc;

 for (idx=0; shl_get(idx, &desc) != -1; idx++) /* step through
libs */
 {
  printf("[%s]\n", desc->filename); /* show imports & exports for
each */
  printf("  Imports:\n");
  num_syms = show_symbols(desc->handle, TYPE_UNDEFINED,
IMPORT_SYMBOLS);
  printf("      TOTAL SYMBOLS: %d\n", num_syms);
  printf("  Exports:\n");
  num_syms = show_symbols(desc->handle, TYPE_UNDEFINED,
EXPORT_SYMBOLS);
  printf("      TOTAL SYMBOLS: %d\n", num_syms);
 }
}
```

The show_all program shown above was compiled with the command:

```
$ cc -Aa -o show_all show_all.c show_symbols.c -ldld
```

NOTE The following output for the example will differ in 64-bit mode. For example, STORAGE is not supported.

The output produced by running this program is shown below:

```
[show_all]
  Imports:
    errno                          STORAGE
    _start                         PROCEDURE
    malloc                         PROCEDURE
    free                           PROCEDURE
    exit                           PROCEDURE
```

```
    printf                          PROCEDURE
    shl_get                         PROCEDURE
    shl_getsymbols                  PROCEDURE
    __d_trap                        PROCEDURE
      TOTAL SYMBOLS: 9
  Exports:
    environ                         DATA       0x40001018
    errno                           STORAGE    0x400011CC
    _SYSTEM_ID                      DATA       0x40001008
    __dld_loc                       STORAGE    0x400011C8
    _FPU_MODEL                      DATA       0x4000100C
    _end                            DATA       0x400011D0
    _environ                        DATA       0x40001018
    __d_trap                        PROCEDURE  0x7AFFF1A6
    main                            PROCEDURE  0x7AFFF1BE
      TOTAL SYMBOLS: 9
[/usr/lib/libc.1]
  Imports:
    _res_rmutex                     STORAGE
    errno                           STORAGE
    _regrpc_rmutex                  STORAGE
    _yellowup_rmutex                STORAGE
    _FPU_MODEL                      STORAGE
    _environ_rmutex                 STORAGE
    _iop_rmutex                     STORAGE
    _rpcnls_rmutex                  STORAGE
    _switch_rmutex                  STORAGE
    _mem_rmutex                     STORAGE
    _dir_rmutex                     STORAGE
```

The shl_unload and cxxshl_unload Routines

Unloads or frees up space for a shared library.

Syntax

int shl_unload(shl_t *handle*)

Parameters

handle The handle of the shared library you wish to unload. The *handle* value is obtained from a previous call to `shl_load`, `shl_findsym`, or `shl_get`.

Return Value

If successful, `shl_unload` returns 0. Otherwise, `shl_unload` returns –1 and sets `errno` to an appropriate value:

`EINVAL` Indicates the specified *handle* is invalid.

Description

To unload a shared library, use the `shl_unload` function. One reason to do this is to free up the private copy of shared library data and swap space allocated when the library was loaded with `shl_load`. (This is done automatically when a process exits.)

Another reason for doing this occurs if a program needs to replace a shared library. For example, suppose you implement some sort of shell or interpreter, and you want to load and execute user “programs” which are actually shared libraries. So you load one program, look up its entry point, and call it. Now you want to run a different program. If you do not unload the old one, its symbol definitions might get in the way of the new library. So you should unload it before loading the new library.

Note that if shared library initializers have been declared for a shared library, they will be called when the shared library is explicitly unloaded. For details, see “Initializers for Shared Libraries”.

If unloading a C++ library, use the `cxxshl_unload` routine. This ensures that destructors of nonlocal static objects are executed when the library is unloaded. The syntax of `cxxshl_unload` is the same as that of `shl_unload`.

Usage

When a library is unloaded, existing linkages to symbols in an unloaded library *are not invalidated*. Therefore, the programmer must ensure that the program does not reference symbols in an unloaded library as undefined behavior will result. In general, this routine is recommended only for experienced programmers.

In 32-bit mode the `shl_unload` routine unloads a shared library irrespective of whether other shared libraries depend on it. In 64-bit mode `shl_unload` unloads a shared library only if no other shared library depend on it.

The dlopen Shared Library Management Routines

This section describes the `dl*` family of shared library management routines.

NOTE Use these routines in 64-bit mode only

The dlopen Routine

Opens a shared library.

Syntax

```
void *dlopen(const char *file, int mode);
```

Parameters

Parm	Definition
file	Used to construct a pathname to the shared library file. If *files* contain a slash character (/), `dlopen` uses the file argument itself as the pathname. If not, `dlopen` searches a series of directories for *file*. • Any directories specified by the environment variable `LD_LIBRARY_PATH`. • Any directories specified by the `RPATH` of the calling load module. • The directories `/usr/lib/pa20_64` and `usr/ccs/lib/pa20_64`.

Parm	Definition	
flags	**Mode**	**Definition**
	RTLD_LAZY	Under this mode, only references to data symbols are relocated when the library t is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode should result in better performance, since a process may not reference all of the functions in any given shared object.
	RTLD_NOW	Under this mode, all necessary relocations are performed when the library is first loaded. This can cause some wasted effort, if relocations are performed for functions that are never referenced, but is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution are available.
	RTLD_GLOBAL	The shared library's symbols are made available for the relocation processing of any other object. In addition, symbol lookup using `dlopen(0, mode)` and an associated `dlsym()` allows objects loaded with RTLD_GLOBAL to be searched.
	RTLD_LOCAL	The shared library's symbols are made available for relocation processing only to objects loaded in the same `dlopen` invocation. If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, the default is RTLD_LOCAL.

Return Values

A successful `dlopen` call returns to the process a *handle* which the process can use on subsequent calls to `dlsym` and `dlclose`. This value should not be interpreted in any way by the process.

`dlopen` returns NULL under the following conditions:

- *file* cannot be found.
- *file* cannot be opened for reading.
- *file* is not a shared object.
- An error occurs during the process of loading *file* or relocating its symbolic references.

More detailed diagnostic information is available through `dlerror`.

Description

`dlopen` is one of a family of routines that give the user direct access to the dynamic linking facilities. `dlopen` makes a shared library specified by a *file* available to a running process. A shared library may specify other objects that it "needs" in order to execute properly. These dependencies are specified by `DT_NEEDED` entries in the.dynamic section of the original shared library. Each needed shared library may, in turn, specify other needed shared libraries. All such shared libraries are loaded along with the original shared library as a result of the call to `dlopen`.

If the value of *file* is 0, `dlopen` provides a *handle* on a "global symbol shared library." This shared library provides access to the symbols from an ordered set of shared libraries consisting of the original `a.out`, all of the shared libraries that were loaded at program startup along with the `a.out`, and all shared libraries loaded using a `dlopen` operation along with the `RTLD_GLOBAL` flag. As the latter set of shared libraries can change during execution, the set identified by *handle* can also change dynamically.

Only a single copy of an shared library file is brought into the address space, even if `dlopen` is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file.

When a shared library is brought into the address space of a process, it can contain references to symbols whose addresses are not known until the shared library is loaded. These references must be relocated before the symbols can be accessed. The mode parameter governs when these relocations take place and may have the following values (defined in Parameters): `RTLD_LAZY` and `RTLD_NOW`.

Any shared library loaded by `dlopen` that requires relocations against global symbols can reference the following:

- Symbols in the original `a.out`.
- Any shared libraries loaded at program startup, from the shared library itself.
- Any shared library included in the same `dlopen` invocation.
- Any shared libraries that were loaded in any `dlopen` invocation that specified the `RTLD_GLOBAL` flag.

To determine the scope of visibility for the symbols loaded with a `dlopen` invocation, bitwise OR the mode parameter with one of the following values: `RTLD_GLOBAL` or `RTLD_LOCAL`.

If neither `RTLD_GLOBAL` nor `RTLD_LOCAL` are specified, the default is `RTLD_LOCAL`.

If a file is specified in multiple `dlopen` invocations, mode is interpreted at each invocation. Note, however, that once `RTLD_NOW` has been specified, the linker operation completes all relocations, rendering any further `RTLD_NOW` operations redundant and any further `RTLD_LAZY` operations irrelevant. Similarly note that once you specify `RTLD_GLOBAL`, the shared library maintains the `RTLD_GLOBAL` status regardless of any previous or future specification of `RTLD_LOCAL`, as long as the shared library remains in the address space [see `dlclose(3C)`].

Symbols introduced into a program through calls to `dlopen` may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous `dlopen` operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to a symbol definition is based on a symbol resolution order. Two such resolution orders are defined: load and dependency ordering.

- Load order establishes an ordering among symbol definitions using the temporal order in which the shared libraries containing the definitions were loaded, such that the definition first loaded has priority over definitions added later. Load ordering is used in relocation processing.
- Dependency ordering uses a “breadth-first” order starting with a given shared library, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied.

The `dlsym` function uses dependency ordering, except when the global symbol shared library is obtained via a `dlopen` operation on *file* with a value 0. The `dlsym` function uses load ordering on the global symbol shared library.

When a `dlopen` operation first makes it accessible, an shared library and its dependent shared libraries are added in dependency order. Once all shared libraries are added, relocations are performed using load order. Note that if an shared library and its dependencies have been loaded by a previous `dlopen` invocation or on startup, the load and dependency order may yield different resolutions.

The symbols introduced by `dlopen` operations and available through `dlsym` are those which are "exported" as symbols of global scope by the shared library. For shared libraries, such symbols are typically those that were specified in (for example) C source code as having `extern` linkage. For `a.out` files, only a subset of externally visible symbols are typically exported: specifically those referenced by the shared libraries with which the `a.out` is linked. The exact set of exported symbols for any shared library or the `a.out` can be controlled using the linker [see `ld(1)`].

NOTE

The environment variable `LD_LIBRARY_PATH` should contain a colon-separated list of directories, in the same format as the `PATH` variable [see `sh(1)`]. `LD_LIBRARY_PATH` is ignored if the process' real user id is different from its effective user id or its real group id is different from its effective group id [see `exec(2)`] or if the process has acquired any privileges [see `tfadmin(1M)`].

Example

The following example shows how to use `dlopen` to load a shared library. The `RTLD_GLOBAL` flag enables global visibility to symbols in `lib1.sl`. The `RTLD_LAZY` flag indicates that only references to data symbols are to be relocated and all function symbol references are to be delayed until their first invocation.

```
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv)
{
    void* handle;

    handle = dlopen("./lib1.sl", RTLD_GLOBAL | RTLD_LAZY);

    if (handle == NULL) {
    printf("Cannot load library\n");
    }
}
```

The dlerror Routine

Gets diagnostic information.

Syntax

```
char *dlerror(void);
```

Description

`dlerror` returns a null-terminated character string (with no trailing newline character) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of `dlerror`, it returns NULL. Thus, invoking `dlerror` a second time, immediately following a prior invocation, results in NULL being returned.

NOTE

The messages returned by `dlerror` may reside in a static buffer that is overwritten on each call to `dlerror`. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message.

Example

The following code sequence shows how to use `dlerror` to get diagnostic information.

```
void*handle;

/* Try to load a non-existing library */
handle = dlopen(“invalid.sl”, RTLD_GLOBAL | RTLD_LAZY);

if (handle == NULL) {
    printf(“%s\n”, dlerror());
}
```

The dlsym Routine

Gets the address of a symbol in shared library.

Syntax

```
void *dlsym(void *handle, const char *name);
```

Parameters

Parameter	Definition
handle	Either the value returned by a call to `dlopen` or the special flag `RTLD_NEXT`. In the former case, the corresponding shared library must not have been closed using `dlclose`.
name	The symbol's name as a character string.

Return Values

If *handle* does not refer to a valid shared library opened by `dlopen`, or if the named symbol cannot be found within any of the shared libraries associated with *handle*, `dlsym` returns NULL. The `dlerror` routine provides more detailed diagnostic information.

Description

`dlsym` allows a process to obtain the address of a symbol defined within a shared library previously opened by `dlopen`.

The `dlsym` routine searches for the named symbol in all shared libraries loaded automatically as a result of loading the shared library referenced by *handle* [see `dlopen(3C)`].

If *handle* is `RTLD_NEXT`, the search begins with the “next” shared library after the shared library from which `dlsym` was invoked. Shared libraries are searched using a load order symbol resolution algorithm [see *dlopen*(3C)]. The “next” shared library, and all other shared libraries searched, are either of global scope (because they were loaded at startup or as part of a `dlopen` operation with the `RTLD_GLOBAL` flag) or are shared libraries loaded by the same `dlopen` operation that loaded the caller of `dlsym`.

Usage

`RTLD_NEXT` can be used to navigate an intentionally created hierarchy of multiply defined symbols created through interposition. For example, if a program wished to create an implementation of `malloc` that embedded some statistics gathering about memory allocations, such an implementation could define its own `malloc` which would gather the

necessary information, and use dlsym with RTLD_NEXT to find the "real" malloc, which would perform the actual memory allocation. Of course, this "real" malloc could be another user-defined interface that added its own value and then used RTLD_NEXT to find the system malloc.

Examples

The following example shows how to use dlopen and dlsym to access either function or data objects. (For simplicity, error checking has been omitted.)

```
void *handle;
int i, *iptr;
int (*fptr)(int);


/* open the needed object */
handle = dlopen("/usr/mydir/mylib.so", RTLD_LAZY);


/* find address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "some_function");


iptr = (int *)dlsym(handle, "int_object");


/* invoke function, passing value of integer as a parameter */


i = (*fptr)(*iptr);
```

The next example shows how to use dlsym with RTLD_NEXT to add functionality to an existing interface. (Error checking has been omitted.)

```
    extern void record_malloc(void *, size_t);


    void *
    malloc(size_t sz)
    {
        void *ptr;
        void *(*real_malloc)(size_t);

        real_malloc = (void * (*) (size_t))
        dlsym(RTLD_NEXT, "malloc");
        ptr = (*real_malloc)(sz);
        record_malloc(ptr, sz);
        return ptr;
    }
```

The dlget Routine

Retrieves information about a loaded module (program or shared library).

Syntax

```
void *dlget(unsigned int index,
    struct load_module_desc *desc,
    size_t desc_size);
```

Parameters

Parameter	Definition
index	Specifies the requested shared library by its placement on the dynamic loader's search list. An index of zero requests information about the program file itself. An index of -1 requests info about the dynamic loader.
desc	Must be preallocated by the user. The structure members are filled in by the dynamic loader with information about the requested shared library.
desc_size	Specifies the size in bytes of the `load_module_desc` structure sent in by the user.

Return Values

If successful, `dlget` returns a handle for the shared library as defined by the return value from `dlopen()`. If a call to `dlget` is unsuccessful, a NULL pointer is returned and *desc* remains unchanged.

Description

`dlget` is one of a family of routines that give the user direct access to the dynamic linking facilities. `dlget` retrieves information about a load module from an index specifying the placement of a load module in the dynamic loader's search list.

A *load_module_desc* structure has the following members:

```
struct load_module_desc {
   unsigned long text_base;
   unsigned long text_size;
   unsigned long data_base;
   unsigned long data_size;
   unsigned long unwind_base;
   unsigned long linkage_ptr;
   unsigned long phdr_base;
   unsigned long tls_size;
   unsigned long tls_start_addr;
   }
```

Example

The following code sequence shows how to use `dlget` to retrieve information about loaded modules. The following code sequence prints the text base of all loaded modules:

```
void*       handle;
int         index;
struct      load_module_desc desc;

for (index = 0; ; i++) {

    handle = dlget(i, &desc, sizeof(struct load_module_desc));

    if (handle = NULL) {
        printf("%s\n", dlerror());
        break;
    }
    else {
        printf("library %d text base = %lx\n", index,
                desc.text_base);
}
}
```

The dlmodinfo Routine

Retrieves information about a loaded module (program or shared library).

Syntax

```
  cc [flag...] file...   -ldl  [library]...

#include    <dlfcn.h>

unsigned long dlmodinfo(unsigned long ip_value,
   struct load_module_desc *desc,
   size_t desc_size,
   void *(*read_tgt_mem)(void* buffer,
      unsigned long ptr,
      size_t bufsiz,
      int ident),
```

```
int ident_parm,
uint64_t load_map_parm);
```

Parameters

Parameter	Description	
ip_value	An address. The instruction pointer value of the requested library.	
desc	A buffer of memory allocated by the user program. The dynamic loader fills this in with module information.	
desc_size	Size in bytes of the *desc* buffer.	
read_tgm_mem	A pointer to a function used by `dlmodinfo` to retrieve needed information. If the value is NULL, the dynamic loader uses its own internal data structures to find the correct load module and ignore the *ident_parm* and *load_map_parm* parameters.	
	buffer	A buffer supplied by `dlmodinfo` to read into.
	ptr	The virtual memory address to read from.
	bufsiz	Tthe size of *buffer* in bytes.
	ident	The value of the *ident_parm* parameter to `dlmodinfo`.
ident_parm	Only used to pass the fourth parameter to *read_tgt_mem*.	
load_map_parm	Only used when calling through *read_tgt_mem*. Contains the starting address of the load map.	

Return Values

If successful, `dlmodinfo` returns a *handle* for the shared library as defined by the return value from `dlopen()`. NULL is returned otherwise. The return values are type-converted to `unsigned long`

Description

`dlmodinfo` is one of a family of routines that give the user direct access to the dynamic linking facilities. The `dlmodinfo` routine retrieves information about a load module from a given address value. `dlmodinfo`

searches all currently loaded load modules looking for a load module whose address range (address range of all loaded segments) holds the given address value. The `dlmodinfo` routine fills the *load_module_desc* with information from the matching load module.

read_tgm_mem allows `dlmodinfo` to find a load module in one process on behalf of another. The calling process passes a callback via *read_tgt_mem* in order to read memory in a different process address space from the one in which `dlmodinfo` resides. *ip_value*, *load_map_parm*, and *ptr* from *read_tgt_mem* can be pointers to shared libraries in another process.

If the calling process calls `dlmodinfo` with a callback registered via *read_tgt_mem*, it must supply the starting address of the target process' load map in the *load_map_parm* parameter to `dlmodinfo`. This can be retrieved from the `DT_HP_LOAD_MAP` entry in the .dynamic section in the target executable file.

Example

The following code sequence shows how to use `dlmodinfo` to retrieve information about a load module. In this example the `dlmodinfo` is provided with the address of a function `foo`. The address of `foo` is matched with the address range (the address range of all loaded segments) of all load modules. The `dlmodinfo` fills in the *load_module_desc* with information form the matching load module.

```
void foo()
{
    printf("foo\n");
}

int retrieve_info()
{
    unsigned      longhandle;

    struct        load_module_desc desc;

    handle = dlmodinfo((unsigned long) &foo,
                       &desc,
                        sizeof(struct load_module_desc),
                        NULL,
                        0,
                         0);

    if (handle != 0) {
    printf("text base = %lx\n", desc.text_base);
    }
}
```

The dlgetname Routine

Retrieves the name of a load module given a load module descriptor.

Syntax

```
char *dlgetname(struct load_module_desc *desc,
size_t desc_size,
void *(*read_tgt_mem)(void* buffer,
      unsigned long long ptr,
      size_t bufsiz,
      int ident),
int ident_parm,
unsigned long long load_map_parm);
```

Parameters

Parameter	Description	
desc	A buffer of memory allocated by the user program. The dynamic loader fills this in with module information.	
desc_size	Size in bytes of the *desc* buffer.	
read_tgm_mem	A pointer to a function used by `dlmodinfo` to retrieve needed information. If the value is NULL, the dynamic loader uses its own internal data structures to find the correct load module and ignore the *ident_parm* and *load_map_parm* parameters.	
	buffer	A buffer supplied by `dlmodinfo` to read into.
	ptr	The virtual memory address to read from.
	bufsiz	The size of *buffer* in bytes.
	ident	The value of the *ident_parm* parameter to `dlmodinfo`.
ident_parm	Only used to pass the fourth parameter to *read_tgt_mem*.	
load_map_parm	Only used when calling through *read_tgt_mem*. Contains the starting address of the load map.	

Return Values

`dlgetname` returns the pathname of a load module represented by *desc*. If *desc* does not describe a loaded module, `dlgetname` returns NULL.

Description

`dlgetname` is one of a family of routines that give the user direct access to the dynamic linking facilities.

The *read_tgt_mem*, *ident_parm*, and *load_map_parm* parameters are identical to those for `dlmodinfo`.

The caller of `dlgetname` must copy the return value to insure that it is not corrupted.

Example

The following code sequence shows how to use `dlgetname` to retrieve the pathname of a load module. This example uses `dlget` to get a *load_module_desc* of the required load module and passes that *load_module_desc* to `dlgetname` to retrieve the pathname.

```
void*handle;
struct      load_module_desc desc;
char*      dll_name;

/* Get load module of the index'th shared library */
handle = dlget(1, &desc, sizeof(struct load_module_desc));

/* Retrieve pathname of the shared library */
dll_name = dlgetname(&desc,
                     sizeof(struct load_module_desc),
                      NULL,
                      0,
                      NULL);
printf("pathname of 1st shared library : %s\n", dll_name);
```

The dlclose Routine

Closes a shared library.

Syntax

```
int dlclose(void *handle);
```

Parameters

Parm	Definition
handle	Value returned by a previous invocation of `dlopen`.

Return Values

If the referenced shared library was successfully closed, `dlclose` returns 0. If the shared library could not be closed, or if *handle* does not refer to an open shared library, `dlclose` returns a non-0 value. More detailed diagnostic information is available through `dlerror`.

Description

`dlclose` disassociates a shared library previously opened by dlopen from the current process. Once an shared library has been closed using `dlclose`, `dlsym` no longer has access to its symbols. All shared libraries loaded automatically as a result of invoking `dlopen` on the referenced shared library [see `dlopen(3C)`] are also closed.

A successful invocation of `dlclose` does not guarantee that the shared libraries associated with *handle* have actually been removed from the address space of the process. shared libraries loaded by one invocation of `dlopen` may also be loaded by another invocation of `dlopen`. The same shared library may also be opened multiple times. An shared library is not removed from the address space until all references to that shared library through an explicit `dlopen` invocation have been closed and all other shared libraries implicitly referencing that shared library have also been closed. Once an shared library has been closed by `dlclose`, referencing symbols contained in that shared library can cause undefined behavior.

Example

The following example shows how to use `dlclose` to unload a shared library:

```
void*    handle;
int        ret_value;

handle = dlopen(“./lib1.sl”, RTLD_GLOBAL | RTLD_LAZY);
```

```
if (handle == NULL) {
    printf("%s\n", dlerror());
}

ret_value = dlclose(handle);

if (ret_value != 0) {
    printf("%s\n", dlerror());
}
```

Dynamic Loader Compatibility Warnings

Starting with the HP-UX 10.20 release, the dynamic loader generates compatibility warnings. These warnings include linker toolset features that may change over time. To display run-time compatibility warnings, set the _HP_DLDOPTS environment variable as follows:

`export _HP_DLDOPTS=-warnings` *Turn on compatibility warnings*

The following sections provide information about the dynamic loader compatibility warnings.

Unsupported Shared Library Management Routines

The following shared library management *shl_load*(3X) routines may become unsupported in a future HP-UX release:

- `shl_definesym()`
- `shl_get()`
- `shl_get_r()`
- `shl_gethandle()`
- `shl_gethandle_r()`
- `shl_getsymbols()`

When these routines become unsupported, the SVR4 *dlopen* (3C) family of routines will be the only dynamic loading routines supported.

Unsupported Shared Library Management Flags

The following shared library management *shl_load*(3X) flags may become unsupported in a future HP-UX-release:

- `BIND_FIRST`
- `BIND_NOSTART`
- `BIND_RESTRICTED`

- `BIND_TOGETHER`
- `BIND_NONFATAL`
- `BIND_VERBOSE`
- `DYNAMIC_PATH`

The following `shl_findsym()` flags may become unsupported in a future release:

- `TYPE_PROCEDURE`
- `TYPE_DATA`
- `TYPE_STORAGE`

NOTE The for HP-UX Release 11.00 64-bit mode linker does not support the `TYPE_STORAGE flag`

7 Position-Independent Code

This chapter discusses

- “What Is Relocatable Object Code?”
- “What is Absolute Object Code?”
- “What Is Position-Independent Code?”
- “Generating Position-Independent Code”

This chapter is useful mainly to programmers who want to write position-independent assembly language code, or who want to convert existing assembly language programs to be position-independent. It is also of interest to compiler developers. This chapter assumes you have a good understanding of virtual memory concepts and memory management.

NOTE

Throughout this chapter, examples of PIC are shown in assembly code.

For the corresponding information for 64-bit mode, see *64-bit Runtime Architecture for PA-RISC 2.0* available from the HP-UX Software Transition Toolkit (STK) at `http://www.software.hp.com/STK/`.

What Is Relocatable Object Code?

Relocatable object code is machine code that is generated by compilers and assemblers and stored in relocatable object files, or `.o` files. A relocatable object file contains symbolic references to locations defined within the compilation unit as well as symbolic references to locations defined outside the compilation unit. The object file also contains relocation information. The linker uses this information to replace the symbolic references with actual addresses.

For example, if you write a program that references the external variable `errno`, the object code created by the compiler contains only a symbolic reference to `errno` since `errno` is not defined in your program. Only when the linker links this object code does the reference to `errno` change (relocate) to an absolute address in virtual memory.

If your program defines a global variable, the compiler assigns a relocatable address to that variable. The compiler also marks all references to that variable as relocatable. The linker replaces the references to the variable with the absolute address of the variable.

What is Absolute Object Code?

Absolute object code is machine code that contains references to actual addresses within the program's address space. When the linker combines relocatable object files to build a program file, or `a.out` file, it writes absolute object code into the file. Thus, when the program is executed, its routines and data must reside at the addresses determined by the linker.

Note that absolute object code *does not contain* **physical addresses**. Physical addresses refer to exact locations in physical memory. Instead, absolute object code contains virtual addresses within a process's address space. These virtual addresses are mapped to physical addresses by the HP-UX virtual memory management system.

Because program files contain absolute virtual addresses, the HP-UX program loader, `exec`, must always load the code and data into the same location within a process's address space. Because this code always resides at the same location within the address space, and because it contains virtual addresses, it is not suitable for shared libraries, although it can be shared by several processes running the same program.

What Is Position-Independent Code?

Position-independent code (**PIC**) is a form of absolute object code that does *not* contain any absolute addresses and therefore does not depend on where it is loaded in the process's virtual address space. This is an important property for building shared libraries.

In order for the object code in a shared library to be fully shareable, it must not depend on its position in the virtual address space of any particular process. The object code in a shared library may be attached at different points in different processes, so it must work independent of being located at any particular position, hence the term position-independent code.

Position independence is achieved by two mechanisms: First, **PC-relative** addressing is used wherever possible for branches within modules. Second, **indirect addressing** through a per-process **linkage table** is used for all accesses to global variables, or for inter-module procedure calls and other branches and literal accesses where PC-relative addressing cannot be used. Global variables must be accessed indirectly since they may be allocated in the main program's address space, and even the relative position of the global variables may vary from one process to another.

The HP-UX dynamic loader (see *dld.sl*(5)) and the virtual memory management system work together to find free space at which to attach position-independent code within a process's address space. The dynamic loader also resolves any virtual addresses that might exist in the library.

Calls to PIC routines are accomplished through a **procedure linkage table** (**PLT**), which is built by the linker. Similarly, references to data are accomplished through a **data linkage table** (**DLT**). Both tables reside in a process's data segment. The dynamic loader fills in these tables with the absolute virtual addresses of the routines and data in a shared library at run time (known as **binding**). Because of this, PIC can be loaded and executed anywhere that a process has free space.

On compilers that support PIC generation, the +z and +Z options cause the compiler to create PIC relocatable object code.

Generating Position-Independent Code

To be position-independent, object code must restrict all references to code and data to either PC-relative or indirect references, where all indirect references are collected in a single linkage table that can be initialized on a per-process basis by `dld.sl`.

Register 19 (`%r19`) is the designated pointer to the linkage table. The linker generates **stubs** that ensure `%r19` always points to the correct value for the target routine and that handle the inter-space calls needed to branch between shared libraries.

The linker generates an **import stub** for each external reference to a routine. The call to the routine is redirected to branch to the import stub, which obtains the target routine address and the new linkage table pointer value from the current linkage table; it then branches to an export stub for the target routine. In 32-bit mode, the linker generates an **export stub** for each externally visible routine in a shared library or program file. The export stub is responsible for trapping the return from the target routine in order to handle the inter-space call required between shared libraries and program files.

NOTE The 64-bit mode linker does not require or support export stubs.

Shown below is the PIC code generated for import and export stubs. Note that this code is generated automatically by the linker; you don't have to generate the stubs yourself.

```
;Import Stub (Incomplete Executable)
X':  ADDIL  L'lt_ptr+ltoff,%dp   ; get procedure entry point
     LDW    R'lt_ptr+ltoff(%r1),%r21
     LDW    R'lt_ptr+ltoff+4(%r1),%r19  ; get new r19 value.
     LDSID  (%r21),%r1
     MTSP   %r1,%sr0
     BE     0(%sr0,%r21)     ; branch to target
     STW    %rp,-24(%sp)     ; save rp

;Import Stub (Shared Library)
X':  ADDIL  L'ltoff,%r19    ; get procedure entry point
     LDW    R'ltoff(%r1),%r21
     LDW    R'ltoff+4(%r1),%r19   ; get new r19 value
     LDSID  (%r21),%r1
     MTSP   %r1,%sr0
     BE     0(%sr0,%r21)     ; branch to target
     STW    %rp,-24(%sp)     ; save rp

;Export Stub (Shared libs and Incomplete Executables)
X':  BL,N   X,%rp ; trap the return
```

```
NOP
LDW    -24(%sp),%rp     ; restore the original rp
LDSID  (%rp),%r1
MTSP   %r1,%sr0
BE,N   0(%sr0,%rp) ; inter-space return
```

For More Information:

The remainder of this section describes how compilers generate PIC for the following addressing situations:

- "PIC Requirements for Compilers and Assembly Code"
- "Long Calls"
- "Long Branches and Switch Tables"
- "Assigned GOTO Statements"
- "Literal References"
- "Global and Static Variable References"
- "Procedure Labels"

You can use these guidelines to write assembly language programs that generate PIC object code. For details on assembly language, refer to the *Assembler Reference Manual* and *PA-RISC 2.0 Architecture*.

PIC Requirements for Compilers and Assembly Code

The linkage table pointer register, `%r19`, must be stored at `%sp`–32 by all PIC routines. This can be done once on procedure entry. `%r19` must also be restored on return from a procedure call. The value should have been stored in `%sp`–32 (and possibly in a callee-saves register). If the PIC routine makes several procedure calls, the routine should copy `%r19` into a callee-saves register as well, to avoid a memory reference when restoring `%r19` upon return from each procedure call. Just like `%r27` (`%dp`), the compilers treat `%r19` as a reserved register whenever PIC mode is in effect.

In general, references to code are handled by the linker, and the compilers act differently only in the few cases where they would have generated long calls or long branches. References to data, however, need a new fixup request to identify indirect references through the linkage table, and the code generated will change slightly.

NOTE Any code which is PIC or which makes calls to PIC must follow the standard procedure call mechanism.

When linking files produced by the assembler, the linker exports only those assembly language routines that have been explicitly exported as `entry` (that is, symbols of type `ST_ENTRY`). Compiler generated assembly code does not explicitly export routines with the `entry` type specified, so the assembly language programmer must ensure that this is done with the `.EXPORT` pseudo-op.

For example: In assembly language, a symbol is exported using

.EXPORT foo, *type*

where *type* can be `code`, `data`, `entry`, and others. To ensure that `foo` is exported from a shared library, the assembly statement must be:

```
.EXPORT foo,entry
```

Long Calls

Normally, the compilers generate a single-instruction call sequence using the `BL` instruction. The compilers can be forced to generate a long call sequence when the module is so large that the `BL` is not guaranteed to reach the beginning of the subspace. In the latter case, the linker can insert a stub. The existing long call sequence is three instructions, using an absolute target address:

```
     LDIL    L'target,%r1
     BLE     R'target(%sr4,%r1)
     COPY    %r1,%rp
```

When the PIC option is in effect, the compilers must generate the following instruction sequence, which is PC-relative:

```
      BL      .+8,%rp                    ; get pc into rp
      ADDIL   L'target - $L0 + 4, %rp    ; add pc-rel offset to rp
      LDO     R'target - $L1 + 8(%r1), %r1
$L0:  LDSID   (%r1), %r31
$L1:  MTSP    %r31, %sr0
      BLE     0(%sr0,%r1)
      COPY    %r31,%rp
```

Long Branches and Switch Tables

Long branches are similar to long calls, but are only two instructions because the return pointer is not needed:

```
     LDIL    L'target,%r1
     BE      R'target(%sr4,%r1)
```

For PIC, these two instructions must be transformed into four instructions, similar to the long call sequence:

```
     BL      .+8,%r1          ; get pc into r1
     ADDIL   L'target-L,%r1   ; add pc-relative offset
L:   LDO     R'target-L,%r1   ; add pc-relative offset
     BV,N    0(%r1)           ; and branch
```

The only problem with this sequence occurs when the long branch is in a switch table, where each switch table entry is restricted to two words. A long branch within a switch table must allocate a linkage table entry and make an indirect branch:

```
     LDW     T'target(%r19),%r1  ; load LT entry
     BV,N    0(%r1)             ; branch indirect
```

Here, the T' operator indicates a new fixup request supported by the linker for linkage table entries.

Assigned GOTO Statements

ASSIGN statements in FORTRAN must be converted to a PC-relative form. The existing sequence forms the absolute address in a register before storing it in the variable:

```
     LDIL    L'target,tmp
     LDO     R'target(tmp),tmp
```

This must be transformed into the following four-instruction sequence:

```
     BL      .+8,tmp          ; get rp into tmp
     DEPI    0,31,2,tmp       ; zero out low-order 2 bits
L:   ADDIL   L'target-L,tmp   ; get pc-rel offset
     LDO     R'target-L(%r1),tmp
```

Literal References

References to literals in the text space are handled exactly like ASSIGN statements (shown above). The LDO instruction can be replaced with LDW as appropriate.

An opportunity for optimization in both cases is to share a single label (L) throughout a procedure, and let the result of BL become a common sub-expression. Thus only the first literal reference within a procedure is expanded to three instructions; the rest remain two instructions.

Global and Static Variable References

References to global or static variables currently require two instructions either to form the address of a variable, or to load or store the contents of the variable:

```
; to form the address of a variable
ADDIL   L'var-$global$+x,%dp
LDO     R'var-$global$+x(%r1),tmp
; to load the contents of a variable
ADDIL   L'var-$global$+x,%dp
LDW     R'var-$global$+x(%r1),tmp
```

These sequences must be converted to equivalent sequences using the linkage table pointer in %r19:

```
; to form the address of a variable
LDW     T'var(%r19),tmp1
LDO     x(tmp1),tmp2    ; omit if x == 0
; to load the contents of a variable
LDW     T'var(%r19),tmp1
LDW     x(tmp1),tmp2
```

Note that the T' fixup on the LDW instruction allows for a 14-bit signed offset, which restricts the DLT to be 16Kb. Because %r19 points to the middle of the DLT, we can take advantage of both positive and negative offsets. The T' fixup specifier should generate a DLT_REL fixup proceeded by an FSEL override fixup. If the FSEL override fixup is not generated, the linker assumes that the fixup mode is LD/RD for DLT_REL fixups. In order to support larger DLT table sizes, the following long form of the above data reference must be generated to reference tables that are larger. If the DLT table grows beyond the 16Kb limit, the linker emits an error indicating that the user must recompile using the +Z option which produces the following long-load sequences for data reference:

```
; form the address of a variable
ADDIL   LT'var,%r19
LDW     RT'var(%r1),tmp1
LDO     x(tmp1),tmp2    ; omit if x == 0
; load the contents of a variable
ADDIL   LT'var,%r19
LDW     RT'var(%r1),tmp1
LDW     x(tmp1),tmp2
```

Procedure Labels

The compilers already mark procedure label constructs so that the linker can process them properly. No changes are needed to the compilers.

When building shared libraries and incomplete executables, the linker modifies the **plabel** calculation (produced by the compilers in both shared libraries and incomplete executables) to load the contents of a DLT entry, which is built for each symbol associated with a CODE_PLABEL fixup.

In shared libraries and incomplete executables, a plabel value is the address of a PLT entry for the target routine, rather than a procedure address; therefore $$dyncall must be used when calling a routine with a procedure label. The linker sets the second-to-last bit in the procedure label to flag this as a special PLT procedure label. The $$dyncall routine checks this bit to determine which type of procedure label has been passed, and calls the target procedure accordingly.

In order to generate a procedure label that can be used for shared libraries and incomplete executables, assembly code must specify that a procedure address is being taken (and that a plabel is wanted) by using the P' assembler fixup mode. For example, to generate an assembly plabel, the following sequence must be used:

```
LDIL LP'function,%r1
LDO RP'function(%r1), %r22
; Now to call the routine
BL $$dyncall, %r31 ; r22 is the input register for $$dyncall
COPY %r31, %r2
```

This code sequence generates the necessary PLABEL fixups that the linker needs in order to generate the proper procedure label. The dyncall millicode routine in /usr/lib/milli.a must be used to call a procedure using this type of procedure label; that is, a BL or BV will not work).

8 Ways to Improve Performance

The linker provides several ways you can improve your application performance.

- “Linker Optimizations” describes how the linker `-O` option removes unnecessary `ADDIL` instructions and “dead” or unused procedures.
- “Options to Improve TLB Hit Rates”describes performance improvements in Translation Lookaside Buffer (TLB) hit rates.
- “Profile-Based Optimization” describes how the linker can position your code in the object file or shared library to improve performance.
- “Improving Shared Library Start-Up Time with fastbind” describes how to improve shared library performance by saving startup information and bypassing the lookup process when running an application.

Linker Optimizations

The linker supports the `-O` option which performs the following optimizations at link time:

- optimizes references to data by removing unnecessary `ADDIL` instructions from the object code.
- removes procedures that can never be reached.

These optimizations can be separately enabled or disabled with the `+O[no]fastaccess` and `+O[no]procelim` options respectively. The `-O` linker option simply combines enabling of these into one option. For example, the following `ld` command enables linker optimizations and results in a smaller, faster executable:

```
$ ld -O -o prog /usr/ccs/lib/crt0.o prog.o -lm -lc
```

To enable one or the other optimization only, use the appropriate `+O` option:

```
$ ld +Ofastaccess -o prog /usr/ccs/lib/crt0.o prog.o -lm -lc
$ ld +Oprocelim -o prog /usr/ccs/lib/crt0.o prog.o -lm -lc
```

Invoking Linker Optimizations from the Compile Line

The compilers automatically call the linker with the `+Ofastaccess` and `+Oprocelim` options *if* compiler optimization level 4 is selected. For example, the following `cc` command invokes full compiler optimization as well as linker optimization:

```
$ cc -o prog +O4 prog.c          O4 invokes +Ofastaccess and +Oprocelim
```

If invoked with `+O4`, the compilers generate object code in such a way that code optimization is done at link time. Thus, the linker does a better job of optimizing code that was compiled with `+O4`.

When the compile and link phases are invoked by separate commands, specify `+O4` on both command lines. For example:

```
$ cc -c +O4 prog.c               invokes compiler optimizations
$ cc -o prog +O4 prog.o          invokes linker optimizations
```

NOTE With the HP-UX 10.0 release, you can also invoke linker optimizations at levels 2 and 3 by using the `+Ofastaccess` or `+Oprocelim` option.

See Also:

For a brief description of compiler optimization options see "Selecting an Optimization Level with PBO". For a complete description, see your compiler manuals or online help.

Incompatibilities with other Options

The `-O`, `+Ofastaccess`, and `+Oprocelim` options are incompatible with these linker options:

`-b`	These options have no effect on position-independent code, so they are not useful when building shared libraries with `ld -b`.
`-A`	Dynamic linking is incompatible with link-time optimization.
`-r`	Relocatable linking is incompatible with link-time optimization.
`-D`	Setting the offset of the data space is incompatible with link-time optimization.

The linker issues a warning when such conflicts occur. If you require any of these features, do not use the linker optimization options.

Unused Procedure Elimination with +Oprocelim

Unused or "dead" procedure elimination is the process of removing unreferenced procedures from the $TEXT$ space of an executable or shared library to reduce the size of the program or library.

Dead procedure elimination is performed after all symbols have been resolved prior to any relocation. It works on a per subspace basis. That is, only entire subspaces are removed and only if all procedures in the subspace are unreferenced. Typically, if a relocatable link (`ld -r`) has not been performed and the code is not written in assembly, every procedure is in its own subspace. Relocatable links may merge

subspaces. Merged subspaces can prevent the removal of dead procedures. Therefore, it is optimal to have each procedure in its own subspace.

Complete Executables

For **complete executables**, dead procedure elimination removes any text subspaces that are not referenced from another subspace. Self references, such as recursive procedures or subspaces with multiple procedures that call each other, are not considered outside references and are therefore candidates for removal.

If the address of a procedure is taken, the subspace within which it resides is not removed. If a subspace is referenced in any way by a fixup representing a reference other than a PC-relative call or an absolute call it is not removed.

Incomplete Executables

For **incomplete executables**, dead procedure elimination works the same as for complete executables except that no exported symbols or their dependencies are removed. If an incomplete executable contains a symbol that is to be referenced by a shared library and is *not* exported, it is removed if the other conditions discussed above hold.

Shared Libraries

In shared libraries only symbols that are not referenced and not exported are removed. In shared libraries all symbols that are not of local scope are exported. Therefore only locally scoped symbols not referenced are removed.

Relocatable Objects

When performing a relocatable link with the `-r` option, dead procedure elimination is disabled since the only possible gain would be the removal of unreferenced local procedures. Objects resulting from a relocatable link are subject to dead procedure elimination upon a final link.

Affects on Symbolic Debugging

Any procedure that has symbolic debug information associated with it is not removed. Procedures that do not have symbolic debug information associated with them but are included in a debug link are removed if they are not referenced.

Options to Improve TLB Hit Rates

To improve Translation Lookaside Buffer (TLB) hit rates in an application running on a PA 8000-based system, use the following linker or `chatr` virtual memory page setting options:

- `+pd` *size* — requests a specified data page *size* of 4K bytes, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, or L. Use `L` to specify the largest page size available. The actual page size may vary if the requested size can not be fulfilled.
- `+pi` *size* — requests a specified instruction page *size*. (See `+pd` *size* for *size* values.)

The default data and instruction page size is 4K bytes on PA-RISC systems.

The PA-RISC 2.0 architecture supports multiple page sizes, from 4K bytes to 64M bytes, in multiples of four. This enables large contiguous regions to be mapped into a single TLB entry. For example, if a contiguous 4MB of memory is actively used, 1000 TLB entries are created if the page size is 4K bytes, but only 64 TLB entries are created if the page size is 64K bytes.

Applications and benchmarks have larger and larger working-set sizes. Therefore, the linker and `chatr` TLB page setting options can help boost performance by improving TLB hit rates.

Some scientific applications benefit from large data pages. Alternatively, some commercial applications benefit from large instruction page sizes.

Examples:

- To set the virtual memory page size by using the linker:

```
ld +pd 64K +pi 16K /opt/langtools/lib/crt0.o myprog.o -lc
```

- To set the page size from HP C and HP Fortran:

```
cc -Wl,+pd,64K,+pi,16K myprog.c
f90 -Wl,+pd,64K,+pi,16K myprog.f
```

- To set the page size by using `chatr`:

```
chatr +pd 64K +pi 16K a.out
```

Profile-Based Optimization

In **profile-based optimization** (PBO), the compiler and linker work together to optimize an application based on profile data obtained from running the application on a typical input data set. For instance, if certain procedures call each other frequently, the linker can place them close together in the `a.out` file, resulting in fewer instruction cache misses, TLB misses, and memory page faults when the program runs. Similar optimizations can be done at the **basic block** levels of a procedure. Profile data is also used by the compiler for other general tasks, such as code scheduling and register allocation.

General Information about PBO

- "When to Use PBO"
- "Restrictions and Limitations of PBO"
- "Compatibility with 9.0 PBO"

Using PBO

- "How to Use PBO"
- "Instrumenting (+I/-I)"
- "Profiling"
- "Optimizing Based on Profile Data (+P/-P)"
- "Selecting an Optimization Level with PBO"
- "Using PBO to Optimize Shared Libraries"
- "Using PBO with ld -r"

NOTE The compiler interface to PBO is currently supported only by the C, C++, and FORTRAN compilers.

When to Use PBO

PBO should be the last level of optimization you use when building an application. As with other optimizations, it should be performed after an application has been completely debugged.

Most applications will benefit from PBO. The two types of applications that will benefit the most from PBO are:

- *Applications that exhibit poor instruction memory locality*. These are usually large applications in which the most common paths of execution are spread across multiple compilation units. The loops in these applications typically contain large numbers of statements, procedure calls, or both.
- *Applications that are branch-intensive*. The operations performed in such applications are highly dependent on the input data. User interface managers, database managers, editors, and compilers are examples of such applications.

Of course, the best way to determine whether PBO will improve an application's performance is to try it.

NOTE Under some conditions, PBO is incompatible with programs that explicitly load shared libraries. Specifically, PBO will not function properly if the `shl_load` routine has either the `BIND_FIRST` or the `BIND_NOSTART` flags set. For more information about explicit loading of shared libraries, see “The shl_load and cxxshl_load Routines” on page 215.

How to Use PBO

Profile-based optimization involves these steps:

1. *Instrument* the application — prepare the application so that it will generate profile data.
2. *Profile* the application — create profile data that can be used to optimize the application.
3. *Optimize* the application — generate optimized code based on the profile data.

A Simple Example

Suppose you want to apply PBO to an application called `sample`. The application is built from a C source file `sample.c`. Discussed below are the steps involved in optimizing the application.

Step 1 Instrumentation

First, compile the application for instrumentation and level 2 optimization:

```
$ cc -v -c +I -O sample.c
/opt/langtools/lbin/cpp sample.c /var/tmp/ctm123
/opt/ansic/lbin/ccom /var/tmp/ctm123 sample.o -O2 -I
$ cc -v -o sample.inst +I -O sample.o
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main \
 -o sample.inst sample.o -I -lc
```

At this point, you have an instrumented program called `sample.inst`.

Step 2 Profile

Assume you have two representative input files to use for profiling, `input.file1` and `input.file2`. Now execute the following three commands:

```
$ sample.inst < input.file1
$ sample.inst < input.file2
$ mv flow.data sample.data
```

The first invocation of `sample.inst` creates the `flow.data` file and places an entry for that executable file in the data file. The second invocation increments the counters for `sample.inst` in the `flow.data` file. The third command moves the `flow.data` file to a file named `sample.data`.

Step 3 Optimize

To perform profile based optimizations on this application, relink the program as follows:

```
$ cc -v -o sample.opt +P +pgm sample.inst \
  +df sample.data sample.o
/usr/ccs/bin/ld /usr/ccs/lib/crt0.o -u main -o sample.opt \
 +pgm sample.inst +df sample.data sample.o -P -lc
```

Note that it was not necessary to recompile the source file. The `+pgm` option was used because the executable name used during instrumentation, `sample.inst`, does not match the current output file name, `sample.opt`. The `+df` option is necessary because the profile database file for the program has been moved from `flow.data` to `sample.data`.

Instrumenting (+I/-I)

Although you can use the linker alone to perform PBO, the best optimizations result if you use the compiler as well; this section describes this approach.

To instrument an application (with C, C++, and FORTRAN), compile the source with the `+I` compiler command line option. This causes the compiler to generate a `.o` file containing intermediate code, rather than the usual object code. (**Intermediate code** is a representation of your code that is lower-level than the source code, but higher level than the object code.) A file containing such intermediate code is referred to as an **I-SOM** file.

After creating an I-SOM file for each source file, the compiler invokes the linker as follows:

1. In 32-bit mode, instead of using the startup file `/usr/ccs/lib/crt0.o`, the compiler specifies a special startup file named `/opt/langtools/lib/icrt0.o`. When building a shared library, the compiler uses `/usr/ccs/lib/scrt0.o`. In 64-bit mode, the linker automatically adds `/usr.css/lib/pa20_64/fdp_init.o` or `/usr.css/lib/pa20_64/fdp_init_sl.o` to the link when detects that `-I crt0.o` is not changed.
2. The compiler passes the `-I` option to the linker, causing it to place instrumentation code in the resulting executable.

You can see how the compiler invokes the linker by specifying the `-v` option. For example, to instrument the file `sample.c`, to name the executable `sample.inst`, to perform level 2 optimizations (the compiler option `-O` is equivalent to `+O2`), and to see verbose output (`-v`):

```
$ cc -v -o sample.inst +I -O sample.c
/opt/langtools/lbin/cpp sample.c /var/tmp/ctm123
/opt/ansic/lbin/ccom /var/tmp/ctm123 sample.o -O2 -I
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main -o \
 sample.inst sample.o -I -lc
```

Notice in the linker command line (starting with `/usr/ccs/bin/ld`), the application is linked with `/opt/langtools/lib/icrt0.o` and the `-I` option is given.

To save the profile data to a file other than `flow.data` in the current working directory, use the `FLOW_DATA` environment variable as described in “Specifying a Different flow.data with FLOW_DATA”.

The Startup File icrt0.o

The `icrt0.o` startup file uses the atexit system call to register the function that writes out profile data. (For 64-bit mode, the initialization code is in `/usr/ccs/lib/pa20_64/fdp_init.0`.) That function is called when the application exits.

`atexit` allows a fixed number of functions to be registered from a user application. Instrumented applications (those linked with `-I`) will have one less `atexit` call available. One or more instrumented shared libraries will use a single additional `atexit` call. Therefore, an instrumented application that contains any number instrumented shared libraries will use two of the available `atexit` calls.

For details on `atexit`, see *atexit*(2).

The -I Linker Option

When invoked with the `-I` option, the linker instruments all the specified object files. Note that the linker instruments regular object files as well as I-SOM files; however, with regular object files, only procedure call instrumentation is added. With I-SOM files, additional instrumentation is done *within* procedures.

For instance, suppose you have a regular object file named `foo.o` created by compiling *without* the `+I` option, and you compile a source file `bar.c` with the `+I` option and specify `foo.o` on the compile line:

```
$ cc -c foo.c
$ cc -v -o foobar -O +I bar.c foo.o
/opt/langtools/lbin/cpp bar.c /var/tmp/ctm456
/opt/ansic/lbin/ccom /var/tmp/ctm456 bar.o -O2 -I
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main -o foobar \
 bar.o foo.o -I -lc
```

In this case, the linker instruments both `bar.o` and `foo.o`. However, since `foo.o` is *not* an I-SOM file, only its procedure calls are instrumented; basic blocks within procedures are not instrumented. To instrument `foo.c` to the same extent, you must compile it with the `+I` option — for example:

```
$ cc -v -c +I -O foo.c
/opt/langtools/lbin/cpp foo.c /var/tmp/ctm432
/opt/ansic/lbin/ccom /var/tmp/ctm432 foo.o -O2 -I
$ cc -v -o foobar -O +I bar.c foo.o
/opt/langtools/lbin/cpp bar.c /var/tmp/ctm456
/opt/ansic/lbin/ccom /var/tmp/ctm456 bar.o -O2 -I
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main -o foobar \
 bar.o foo.o -I -lc
```

A simpler approach would be to compile `foo.c` and `bar.c` with a single `cc` command:

```
$ cc -v +I -O -o foobar bar.c foo.c
/opt/langtools/lbin/cpp bar.c /var/tmp/ctm352
/opt/ansic/lbin/ccom /var/tmp/ctm352 bar.o -O2 -I
/opt/langtools/lbin/cpp foo.c /var/tmp/ctm456
/opt/ansic/lbin/ccom /var/tmp/ctm456 foo.o -O2 -I
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main -o foobar \
 bar.o foo.o -I -lc
```

Code Generation from I-SOMs

As discussed in "Looking "inside" a Compiler" on page 38, a compiler driver invokes several phases. The last phase before linking is **code generation**. When using PBO, the compilation process stops at an intermediate code level. The PA-RISC code generation and optimization phase is invoked by the linker. The code generator is `/opt/langtools/lbin/ucomp`.

NOTE Since the code generation phase is delayed until link time with PBO, linking can take much longer than usual when using PBO. Compile times are faster than usual, since code generation is not performed.

Profiling

After instrumenting a program, you can run it one or more times to generate profile data, which is ultimately used to perform the optimizations in the final step of PBO.

This section provides information on the following profiling topics:

- "Choosing Input Data"
- "The flow.data File"
- "Storing Profile Information for Multiple Programs"
- "Sharing the flow.data File Among Multiple Processes"
- "Forking an Instrumented Application"

Choosing Input Data

For best results from PBO, use representative input data when running an instrumented program. Input data that represents rare cases or error conditions is usually not effective for profiling. Run the instrumented program with input data that closely resembles the data in a typical

user's environment. Then, the optimizer will focus its efforts on the parts of the program that are critical to performance in the user's environment. You should not have to do a large number of profiling runs before the optimization phase. Usually it is adequate to select a small number of representative input data sets.

The flow.data File

When an instrumented program terminates with the *exit*(2) system call, special code in the 32-bit `icrt0.o` startup file or the 64-bit `/usr/ccs/lib/pa20_64/fdp_init.o` file writes profile data to a file called `flow.data` in the current working directory. This file contains binary data, which cannot be viewed or updated with a text editor. The `flow.data` file is not updated when a process terminates without calling `exit`. That happens, for example, when a process aborts due to an unexpected signal, or when program calls *exec*(2) to replace itself with another program.

There are also certain non-terminating processes (such as servers, daemons, and operating systems) which never call `exit`. For these processes, you must programmatically write the profile data to the `flow.data` file. In order to do so, a process must call a routine called `_write_counters()`. This routine is defined in the `icrt0.o` file. A stub routine with the same name is present in the `crt0.o` file so that the source does not have to change when instrumentation is not being done.

If `flow.data` does not exist, the program creates it; if `flow.data` exists, the program updates the profile data.

As an example, suppose you have an instrumented program named `prog.inst`, and two representative input data files named `input_file1` and `input_file2`. Then the following lines would create a `flow.data` file:

```
$ prog.inst < input_file1
$ ls flow.data
  flow.data
$ prog.inst < input_file2
```

The `flow.data` file includes profile data from both input files.

To save the profile data to a file other than `flow.data` in the current working directory, use the `FLOW_DATA` environment variable as described in “Specifying a Different flow.data with FLOW_DATA”.

Storing Profile Information for Multiple Programs

A single `flow.data` file can store information for multiple programs. This allows an instrumented program to spawn other instrumented programs, all of which share the same `flow.data` file.

To allow multiple programs to save their data in the same `flow.data` file, a program's profile data is uniquely identified by the executable's basename (see *basename*(1)), the executable's file size, and the time the executable was last modified.

Instead of using the executable's basename, you can specify a basename by setting the environment variable `PBO_PGM_PATH`. This is useful when a number of programs are actually linked to the same instrumented executables.

For example, consider profiling the `ls`, `lsf` and `lsx` commands. (`lsx` is `ls` with the `-x` option and `lsf` is `ls` with the `-F` option.) Since the three commands could be linked to the same instrumented executables, the developer might want to collect profile data under a single basename by setting `PBO_PGM_PATH=ls`. If `PBO_PGM_PATH=ls` were not set, profile data would be saved under the `ls`, the `lsf`, and the `lsx` basenames.

When an instrumented program begins execution, it checks whether the basename, size, and time-stamp match those in the existing `flow.data` file. If the basename matches but the size or time-stamp does not match, that probably means that the program has been relinked since it last created profile data. In this case, the following error message will be issued:

```
program: Can't update counters.  Profile data exists
         but does not correspond to this executable.  Exit.
```

You can fix this problem any one of these ways:

- Remove or rename the existing `flow.data` file.
- Run the instrumented program in a different working directory.
- Set the `FLOW_DATA` environment variable so that profile data is written to a file other than `flow.data`.
- Rename the instrumented program.

Sharing the flow.data File Among Multiple Processes

A `flow.data` file can potentially be accessed by several processes at the same time. For example, this could happen when you run more than one instrumented program at the same time in the same directory, or when profiling one program while linking another with `-P`.

Such asynchronous access to the file could potentially corrupt the data. To prevent simultaneous access to the `flow.data` file in a particular directory, a **lock file** called `flow.lock` is used. Instrumented programs that need to update the `flow.data` file and linker processes that need to read it must first obtain access to the lock file. Only one process can hold the lock at any time. As long as the `flow.data` file is being actively read and written, a process will wait for the lock to become available.

A program that terminates abnormally may leave the `flow.data` file inactive but locked. A process that tries to access an inactive but locked `flow.data` file gives up after a short period of time. In such cases, you may need to remove the `flow.lock` file.

If an instrumented program fails to obtain the database lock, it writes the profile data to a temporary file and displays a warning message containing the name of the file. You could then use the `+df` option along with the `+P` option while optimizing, to specify the name of the temporary file instead of the `flow.data` file.

If the linker fails to obtain the lock, it displays an error message and terminates. In such cases, wait until all active processes that are reading or writing a profile database file in that directory have completed. If no such processes exist, remove the `flow.lock` file.

Forking an Instrumented Application

When instrumenting an application that creates a copy of itself with the `fork` system call, you must ensure that the child process calls a special function named `_clear_counters()`, which clears all internal profile data. If you don't do this, the child process inherits the parent's profile data, updating the data as it executes, resulting in inaccurate (exaggerated) profile data when the child terminates. The following code segment shows a valid way to call `_clear_counters`:

```
if ((pid = fork()) == 0) /* this is the child process */
{
  _clear_counters();     /* reset profile data for child */
       . . .             /* other code for the child */
}
```

The function `_clear_counters` is defined in `icrt0.o`. It is also defined as a stub (an empty function that does nothing) in `crt0.o`. This allows you to use the same source code without modification in the instrumented and un-instrumented versions of the program.

Optimizing Based on Profile Data (+P/-P)

The final step in PBO is optimizing a program using profile data created in the profiling phase. To do this, rebuild the program with the `+P` compiler option. As with the `+I` option, the `+P` option causes the compiler to generate an I-SOM `.o` file, rather than the usual object code, for each source file.

Note that it is not really necessary to recompile the source files; you could, instead, specify the I-SOM `.o` files that were created during the instrumentation phase. For instance, suppose you have already created an I-SOM file named `foo.o` from `foo.c` using the `+I` compiler option; then the following commands are equivalent in effect:

```
cc +P foo.c
cc +P foo.o
```

Both commands invoke the linker, but the second command doesn't compile before invoking the linker.

The -P Linker Option

After creating an I-SOM file for each source file, the compiler driver invokes the linker with the `-P` option, causing the linker to optimize all the `.o` files. As with the `+I` option, the driver uses `/opt/langtools/lbin/ucomp` to generate code and perform various optimizations.

To see how the compiler invokes the linker, specify the `-v` option when compiling. For instance, suppose you have instrumented `prog.c` and gathered profile data into `flow.data`. The following example shows how the compiler driver invokes the linker when `+P` is specified:

```
$ cc -o prog -v +P prog.o
/usr/ccs/bin/ld /usr/ccs/lib/crt0.o -u main -o prog \
 prog.o -P -lc
```

Notice how the program is now linked with `/usr/ccs/lib/crt0.o` instead of `/opt/langtools/lib/icrt0.o` because the profiling code is no longer needed.

Using The flow.data File

By default, the code generator and linker look for the `flow.data` file in the current working directory. In other words, the `flow.data` file created during the profiling phase should be located in the directory where you relink the program.

Specifying a Different flow.data File with +df

What if you want to use a `flow.data` file from a different directory than where you are linking? Or what if you have renamed the `flow.data` file — for example, if you have multiple `flow.data` files created for different input sets? The `+df` option allows you to override the default `+P` behavior of using the file `flow.data` in the current directory. The compiler passes this option directly to the linker.

For example, suppose after collecting profile data, you decide to rename `flow.data` to `prog.prf`. You could then use the `+df` option as follows:

```
$ cc -v -o prog +P +df prog.prf prog.o
/usr/ccs/bin/ld /usr/ccs/lib/crt0.o -u main -o prog \
 +df prog.prf prog.o -P -lc
```

The `+df` option overrides the effects of the `FLOW_DATA` environment variable.

Specifying a Different flow.data with FLOW_DATA

The `FLOW_DATA` environment variable provides another way to override the default `flow.data` file name and location. If set, this variable defines an alternate file name for the profile data file.

For example, to use the file `/home/adam/projX/prog.data` instead of `flow.data`, set `FLOW_DATA`:

```
$ FLOW_DATA=/home/adam/projX/prog.data
$ export FLOW_DATA                              Bourne and Korn shell
$ setenv FLOW_DATA /home/adam/projX/prog.data   C shell
```

Interaction between FLOW_DATA and +df

If an application is linked with `+df` and `-P`, the `FLOW_DATA` environment variable is ignored. In other words, `+df` overrides the effects of `FLOW_DATA`.

Specifying a Different Program Name (+pgm)

When retrieving a program's profile data from the `flow.data` file, the linker uses the program's basename as a lookup key. For instance, if a program were compiled as follows, the linker would look for the profile data under the name `foobar`:

```
$ cc -v -o foobar +P foo.o bar.o
/usr/ccs/bin/ld /usr/ccs/lib/crt0.o -u main -o foobar \
 foo.o bar.o -P -lc
```

This works fine as long as the name of the program is the same during the instrumentation and optimization phases. But what if the name of the instrumented program is not the same as name of the final optimized program? For example, what if you want the name of the instrumented application to be different from the optimized application, so you use the following compiler commands?

```
$ cc -O +I -o prog.inst prog.c        Instrument prog.inst.
$ prog.inst < input_file1             Profile it, storing the data
                                      under the name prog.inst.
$ prog.inst < input_file2
$ cc +P -o prog.opt prog.c            Optimize it, but name it prog.opt.
```

The linker would be unable to find the program name `prog.opt` in the `flow.data` file and would issue the error message:

```
No profile data found for the program prog.opt in flow.data
```

To get around this problem, the compilers and linker provide the `+pgm` *name* option, which allows you to specify a program name to look for in the `flow.data` file. For instance, to make the above example work properly, you would include `+pgm prog.inst` on the final compile line:

```
$ cc +P -o prog.opt +pgm prog.inst prog.c
```

Like the `+df` option, the `+pgm` option is passed directly to the linker.

Selecting an Optimization Level with PBO

When `-P` is specified, the code generator and linker perform profile-based optimizations on any I-SOM or regular object files found on the linker command line. In addition, optimizations will be performed according to the optimization level you specified with a compiler option when you instrumented the application. Briefly, the compiler optimization options are:

`+O0`	Minimal optimization. This is the default.
`+O1`	Basic block level optimization.

+02	Full optimization within each procedure in a file. (Can also be invoked as -O.)
+03	Full optimization across all procedures in an object file. Includes subprogram inlining.
+04	Full optimization across entire application, performed at link time. (Invokes `ld +Ofastaccess +Oprocelim`.) Includes inlining across multiple files.

NOTE

Be aware that +03 and +04 are incompatible with symbolic debugging. The only compiler optimization levels that allow for symbolic debugging are +02 and lower.

For more detailed information on compiler optimization levels, see your compiler documentation.

PBO has the greatest impact when it is combined with level 2 or greater optimizations. For instance, this compile command combines level 2 optimization with PBO (note that the compiler options +02 and -O are equivalent):

```
$ cc -v -O +I -c prog.c
/opt/langtools/lbin/cpp prog.c /var/tmp/ctm123
/opt/ansic/lbin/ccom /var/tmp/ctm123 prog.o -O2 -I
$ cc -v -O +I -o prog prog.o
/usr/ccs/bin/ld /opt/langtools/lib/icrt0.o -u main -o prog \
 prog.o -I -lc
```

The optimizations are performed along with instrumentation. However, profile-based optimizations are not performed until you compile later with +P:

```
$ cc -v +P -o prog prog.o
/usr/ccs/bin/ld /usr/ccs/lib/crt0.o -u main \
     -o prog prog.o -P -lc
```

Using PBO to Optimize Shared Libraries

Beginning with the HP-UX 10.0 release, the -I linker option can be used with -b to build a shared library with instrumented code. Also, the -P, +df, and +pgm command-line options are compatible with the -b option.

To profile shared libraries, you must set the environment variable SHLIB_FLOW_DATA to the file that receives profile data. Unlike FLOW_DATA, SHLIB_FLOW_DATA has no default output file. If SHLIB_FLOW_DATA is not set, profile data is not collected. This allows you to activate or suspend the profiling of instrumented shared libraries.

Note that you could set SHLIB_FLOW_DATA to `flow.data` which is the same file as the default setting for FLOW_DATA. But, again, profile data will not be collected from shared libraries unless you explicitly set SHLIB_FLOW_DATA to some output file.

The following is a simple example for instrumenting, profiling, and optimizing a shared library:

```
$ cc +z +I -c -O libcode.c              Create I-SOM files.
$ ld -b -I libcode.o -o mylib.inst.sl   Create instrumented sl.
$ cc main.c mylib.inst.sl               Creat executablea.outile.
$ export SHLIB_FLOW_DATA=./flow.data    Specify output file for
                    profile data
$ a.out < input_file                    Run instrumented executable
                                        with representative input data

$ ld -b -P +pgm mylib.inst.sl \
  libcode.o -o mylib.sl                 Perform PBO.
```

Note that the name used in the database will be the output pathname specified when the instrumented library is linked (`mylib.inst.sl` in the example above), regardless of how the library might be moved or renamed after it is created.

Using PBO with ld -r

Beginning with the HP-UX 10.0 release, you can take greater advantage of PBO on merged object files created with the `-r` linker option.

Briefly, `ld -r` combines multiple `.o` files into a single `.o` file. It is often used in large product builds to combine objects into more manageable units. It is also often used in combination with the linker `-h` option to hide symbols that may conflict with other subsystems in a large application. (See “Hiding Symbols with -h” on page 81 for more information on `ld -h`.)

In HP-UX 10.0, the subspaces in the merged `.o` file produced by `ld -r` are relocatable which allows for greater optimization.

The following is a simple example of using PBO with `ld -r`:

```
$ cc +I -c file1.c file2.c              Create individual I-SOM files
$ ld -r -I -o reloc.o file1.o file2.o   Build relocatable, merged file.
$ cc +I -o a.out reloc.o                Create instrumented executable file.
$ a.out < input_file                    Run instrumented executable with
                                        representative input data.


$ ld -r -P +pgm a.out -o reloc.o \
  file1.o file2.o                       Rebuild relocatable file for PBO.
$ cc +P -o a.out reloc.o                Perform PBO on the final executable file.
```

Notice in the example above, that the `+pgm` option was necessary because the output file name differs from the instrumented program file name.

NOTE

If you are using `-r` and C++ templates, check "Known Limitations" in the *HP C++ Release Notes* for possible limitations.

Restrictions and Limitations of PBO

This section describes restrictions and limitations you should be aware of when using Profile-Based Optimization.

- "Temporary Files"
- "Source Code Changes and PBO"
- "Profile-Based Optimization (PBO) and High-Level Optimization (HLO)"
- "I-SOM File Restrictions"

PBO calls `malloc()` during the instrumentation (`+I`) phase. If you replace `libc` *malloc*(3C) calls with your own version of `malloc()`, use the same parameter list (data types, order, number, and meaning of parameters) as the HP version. (For information on `malloc()`, see *malloc*(3C).)

Temporary Files

The linker does not modify I-SOM files. Rather, it compiles, instruments, and optimizes the code, placing the resulting temporary object file in a directory specified by the `TMPDIR` environment variable. If PBO fails due to inadequate disk space, try freeing up space on the disk that contains the `$TMPDIR` directory. Or, set `TMPDIR` to a directory on a disk with more free space.

Source Code Changes and PBO

To avoid the potential problems described below, PBO should only be used during the final stages of application development and performance tuning, when source code changes are the least likely to be made. Whenever possible, an application should be re-profiled after source code changes have been made.

What happens if you attempt to optimize a program using profile data that is older than the source files? For example, this could occur if you change source code and recompile with +P, but don't gather new profile data by re-instrumenting the code.

In that sequence of events, optimizations will still be performed. However, full profile-based optimizations will be performed only on those procedures whose internal structure has not changed since the profile data was gathered. For procedures whose structure *has* changed, the following warning message is generated:

```
ucomp warning: Code for name changed since profile
database file flow.data built.  Profile data for name
ignored.  Consider rebuilding flow.data.
```

Note that it is possible to make a source code change that does not affect the control flow structure of a procedure, but which does significantly affect the profiling data generated for the program. In other words, a very small source code change can dramatically affect the paths through the program that are most likely to be taken. For example, changing the value of a program constant that is used as a parameter or loop limit value might have this effect. If the user does not re-profile the application after making source code changes, the profile data in the database will not reflect the effects of those changes. Consequently, the transformations made by the optimizer could degrade the performance of the application.

Profile-Based Optimization (PBO) and High-Level Optimization (HLO)

High-level optimization, or HLO, consists of a number of optimizations, including inlining, that are automatically invoked with the +O3 and +O4 compiler options. (Inlining is an optimization that replaces each call to a routine with a copy of the routine's actual code.) +O3 performs HLO on each module while +O4 performs HLO over the entire program and removes unnecessary ADDIL instructions. Since HLO distorts profile data, it is suppressed during the instrumentation phases of PBO.

When +I is specified along with +O3 or +O4, an I-SOM file is generated. However, HLO is not performed during I-SOM generation. When the I-SOM file is linked, using the +P option to do PBO, HLO is performed, taking advantage of the profile data.

Example . The following example illustrates high-level optimization with PBO:

```
$ cc +I +O3 -c file.c    Create I-SOM for instrumentation.
$ cc +I +O3 file.o       Link with instrumentation.
$ a.out < input_file     Run instrumented executable with representative input data.
$ cc +P +O3 file.o       Perform PBO and HLO.
```

Replace +O3 with +O4 in the above example to get HLO over the entire program and ADDIL elimination. (You may see a warning when using +O4 at instrumentation indicating that the +O4 option is being ignored. You can ignore this warning.)

I-SOM File Restrictions

For the most part, there are not many noticeable differences between I-SOM files and ordinary object files. Exceptions are noted below.

ld . Linking object files compiled with the +I or +P option takes much longer than linking ordinary object files. This is because in addition to the work that the linker already does, the code generator must be run on the intermediate code in the I-SOM files. On the other hand, the time to compile a file with +I or +P is relatively fast since code generation is delayed until link time.

All options to ld should work normally with I-SOM files with the following exceptions:

-r	The -r option works with both -I and -P. However, it produces an object file, *not* an I-SOM file. In 64-bit mode, use -I, -P, or the +nosectionmerge option on a -r linker command to allow procedures to be positioned independently. Without these options, a -r link merges procedures into a single section.
-s	Do not use this option with -I. However, there is no problem using this option with -P.
-G	Do not use this option with -I. There is no problem using this option with -P.
-A	Do not use this option with -I or -P.
-N	Do not use this option with -I or -P.

nm . The `nm` command works on I-SOM files. However, since code generation has not yet been performed, some of the imported symbols that might appear in an ordinary relocatable object file will not appear in an I-SOM file.

ar . I-SOM files can be manipulated with `ar` in exactly the same way that ordinary relocatable files can be.

strip . Do not run `strip` on files compiled with `+I` or `+P`. Doing so results in an object file that is essentially empty.

Compiler Options . Except as noted below, all `cc`, `CC`, and `f77` compiler options work as expected when specified with `+I` or `+P`:

`-g`	This option is incompatible with `+I` and `+P`.
`-G`	This option is incompatible with `+I`, but compatible with `+P` (as long as the insertion of the `gprof` library calls does not affect the control flow graph structure of the procedures.)
`-p`	This option is incompatible with `+I` option, but is compatible with `+P` (as long as the insertion of the `prof` code does not affect the control flow graph structure of the procedures.)
`-s`	You should not use this option together with `+I`. Doing so will result in an object file that is essentially empty.
`-S`	This option is incompatible with `+I` and `+P` options because assembly code is not generated from the compiler in these situations. Currently, it is not possible to get assembly code listings of code generated by `+I` and `+P`.
`-y/+y`	The same restrictions apply to these options that were mentioned for `-g` above.
`+o`	This option is incompatible with `+I` and `+P`. Currently, you cannot get code offset listings for code generated by `+I` and `+P`.

Compatibility with 9.0 PBO

PBO is largely compatible between the 9.0 and 10.0 releases.

I-SOM files created under 9.0 are completely acceptable in the 10.0 environment.

However, it is advantageous to re-profile programs under 10.0 in order to achieve improved optimization. Although you can use profile data in `flow.data` files created under 9.0, the resulting optimization will not take advantage of 10.0 enhancements. In addition, a warning is generated stating that the profile data is from a previous release. See the section called “Profiling” in this chapter for more information.

See the section called “Profiling” for more information about the warning generated for profile data generated from a previous release.

Improving Shared Library Start-Up Time with fastbind

The `fastbind` tool improves the start-up time of programs that use shared libraries. When `fastbind` is invoked, it caches **relocation** information inside the executable file. The next time the executable file runs, the dynamic loader uses this cached information to bind the executable instead of searching for symbols.

The syntax for fastbind is:

`fastbind` [-n] [-u] *incomplete executable...*

where:

-n	Removes `fastbind` data from the executable.
-u	Performs `fastbind` even when unresolved symbols are found. (By default, fastbind stops when it cannot resolve symbols.)

Using fastbind

You can create and delete `fastbind` information for an executable file after it has been linked with shared libraries. You can invoke `fastbind` from the linker or use the `fastbind` tool directly. You can set the `_HP_DLDOPTS` environment variable to find out if `fastbind` information is out-of-date and to turn off `fastbind` at run time.

Invoking the fastbind Tool

To invoke `fastbind` on an **incomplete executable** file, verify that your executable has write access (because fastbind writes to the file) and then run `fastbind`.

```
$ ls -l main
-rwxrwxrwx   1 janet     191          28722 Feb 20 09:11 main
$ fastbind main
```

The `fastbind` tool generates `fastbind` information for `main` and rewrites `main` to contain this information.

Invoking fastbind from the Linker

To invoke fastbind from `ld`, pass the request to the linker from your compiler by using the `-Wl,+fb` options. For example:

```
$ ld -b convert.o volume.o -o libunits.sl     Build the shared library.
$ cc -Aa -Wl,+fb main.c -o main \             Link main to the shared
  libunits.sl -lc                             library.  Perform fastbind.
```

The linker performs `fastbind` after it creates the executable file.

How to Tell if fastbind Information is Current

By default, when the dynamic loader finds that `fastbind` information is out-of-date, it silently reverts back to the standard method for binding symbols. To find out if an executable file has out-of-date fastbind information, set the `_HP_DLDOPTS` environment variable as follows:

```
$ export _HP_DLDOPTS=-fbverbose
$ main
/usr/lib/dld.sl: Fastbind data is out of date
```

The dynamic loader provides a warning when the `fastbind` information is out-of-date.

Removing fastbind Information from a File

To remove `fastbind` information from a file, use the `fastbind` tool with the `-n` option. For example:

```
$ fastbind -n main                    Remove fastbind information from main.
```

Turning off fastbind at Run Time

To use the standard search method for binding symbols, instead of the `fastbind` information in an executable file, set the `_HP_DLDOPTS` environment variable as follows:

```
export _HP_DLDOPTS=-nofastbind        Turns off fastbind at run time.
```

For More Information:

See the *fastbind*(1) man page.

A Using Mapfiles

The `ld` command automatically maps sections from input object files onto output segments in executable files. The mapfile option allows you to change the default mapping provided by the linker.

NOTE The mapfile option is supported only in 64-bit mode linking.

NOTE In most cases, the linker produces a correct executable without the use of the mapfile option. The mapfile option is an advanced feature of the linker toolset intended for system programming use, not for application programming use. When using the mapfile option, you can easily create executable files that do not execute.

Controlling Mapfiles with the -k Option

The `-k` option to `ld` specifies a text file containing mapfile directives:

`ld -k` *mapfile* [*flags*] *files* `...`

The `ld` command automatically maps sections from input object files onto output segments in executable files. The mapfile option allows you to change the default mapping provided by the linker.

Use the `-k` *filename* option to specify a text file that contains mapfile directives. The linker appends the specified mapfile to the default mapfile unless you specify the `+nodefaultmap` option.

Mapfile Example: Using -k filename (without +nodefaultmap Option):

`cat mapfile`

```
text = LOAD ?RX V0x1000;
text : .rodata;
text : .PARISC.milli;
text : .dynamic;
text : .dynsym;
text : .dynstr;
text : .hash;
text : $PROGBITS ?AX;
text : .PARISC.unwind;
text : $UNWIND;
data = LOAD ?RW V0x4000000040001000;
data : .opd;
data : .plt;
data : .dlt;
data : .data;
data : $PROGBITS ?AW!S;
data : .sdata;
data : $PROGBITS ?AWS;
data : .sbss;
data : $NOBITS ?AWS;
data : .bss;
data : $NOBITS ?AW!S;
note = NOTE;
note : $NOTE;
```

`ld main.o -k mapfile -lc`

`elfdump -h -S a.out`

```
a.out:

*** Section Header ***

Index TypeVaddr           Offset           Size     Name
```

```
1  DYNM 00000000000012a8  00000000000002a8  00000120 .dynamic
1  DYNM 00000000000011c8  00000000000001c8  00000120 .dynamic
2  DYNS 00000000000012e8  00000000000002e8  00000270 .dynsym
3  STRT 0000000000001558  0000000000000558  00000113 .dynstr
4  HASH 0000000000001670  0000000000000670  000000a4 .hash
5  PBIT 0000000000001718  0000000000000718  00000044 .text
6  PBIT 000000000000175c  000000000000075c  00000018 .interp
10 RELA 0000000000001778  0000000000000778  00000000 .rela.opd
15 PBIT 4000000040001020  0000000000001020  00000010 .plt
16 PBIT 4000000040001030  0000000000001030  00000000 .dlt
17 PBIT 4000000040001030  0000000000001030  00000000 .data
18 PBIT 4000000040001030  0000000000001030  00000000 .HP.init
19 PBIT 4000000040001030  0000000000001030  00000000 .preinit
20 PBIT 4000000040001030  0000000000001030  00000000 .init
21 PBIT 4000000040001030  0000000000001030  00000000 .fini
22 PBIT 4000000040001030  0000000000001030  00000008 .sdata
23 NOBI 4000000040001038  0000000000001038  00000008 .bss
24 NOBI 0000000000000000  0000000000001038  00000000 .tbss
25 STRT 0000000000000000  0000000000001038  000001b2 .strtab
26 SYMT 0000000000000000  00000000000011ec  000004b0 .symtab
27 STRT 0000000000000000  000000000000169c  000000de .shstrtab
```

Changing Mapfiles with -k filename and +nodefaultmap

The +nodefaultmap option used with -k option prevents the linker from concatenating the default memory map to the map provided by *filename*. If you specify +nodefaultmap, the linker does not append the default mapfile to your mapfile. If you do not specify +nodefaultmap with -k, the linker appends the default to the output file.

Mapfile Example: Using -k *mapfile* and +nodefaultmap

cat mapfile

```
text = LOAD ?RX V0x1000;
text : .rodata;
text : .PARISC.milli;
text : .dynamic;
text : .dynsym;
text : .dynstr;
text : .hash;
text : $PROGBITS ?AX;
text : .PARISC.unwind;
text : $UNWIND;
data = LOAD ?RW V0x4000000040001000;
data : .opd;
data : .plt;
data : .dlt;
data : .data;
data : $PROGBITS ?AW!S;
data : .sdata;
data : $PROGBITS ?AWS;
data : .sbss;
data : $NOBITS ?AWS;
data : .bss;
data : $NOBITS ?AW!S;
note = NOTE;
note : $NOTE;
```

ld main.o +nomapfile -k mapfile -lc

elfdump -h -S a.out

```
a.out:

*** Section Header ***

Index TypeVaddr          Offset           Size     Name
1 DYNM  00000000000011c8  00000000000001c8  00000120 .dynamic
2 DYNS  00000000000012e8  00000000000002e8  00000270 .dynsym
3 STRT  0000000000001558  0000000000000558  00000113 .dynstr
4 HASH  0000000000001670  0000000000000670  000000a4 .hash
```

```
 5 PBIT 0000000000001718  0000000000000718  00000044 .text
 6 PBIT 000000000000175c  000000000000075c  00000018 .interp
 7 RELA 0000000000001778  0000000000000778  00000000 .rela.HP.init
 8 RELA 0000000000001778  0000000000000778  00000000 .rela.init
 9 RELA 0000000000001778  0000000000000778  00000000 .rela.fini
10RELA 0000000000001778  0000000000000778  00000000 .rela.opd
11RELA 0000000000001778  0000000000000778  00000018 .rela.plt
12RELA 0000000000001790  0000000000000790  00000000 .rela.dlt
13UNWI 0000000000001790  0000000000000790  00000010 .PARISC.unwind
14PBIT 4000000040001000  0000000000001000  00000020 .opd
15PBIT 4000000040001020  0000000000001020  00000010 .plt
16PBIT 4000000040001030  0000000000001030  00000000 .dlt
17PBIT 4000000040001030  0000000000001030  00000000 .data
18PBIT 4000000040001030  0000000000001030  00000000 .HP.init
19PBIT 4000000040001030  0000000000001030  00000000 .preinit
20PBIT 4000000040001030  0000000000001030  00000000 .init
21PBIT 4000000040001030  0000000000001030  00000000 .fini
22PBIT 4000000040001030  0000000000001030  00000008 .sdata
23NOBI 4000000040001038  0000000000001038  00000008 .bss
24NOBI 0000000000000000  0000000000001038  00000000 .tbss
25STRT 0000000000000000  0000000000001038  000001b2 .strtab
26SYMT 0000000000000000  00000000000011ec  000004b0 .symtab
27STRT 0000000000000000  000000000000169c  000000de .shstrtab
```

Simple Mapfile

The following directives show how a simple mapfile would appear:

```
# text segment
text = LOAD ?RX;
text : .rodata ?A;
text : $PROGBITS ?AX;
# data segment
data = LOAD ?RW;
data : $PROGBITS ?AW!S;
data : $PROGBITS ?AWS;
data : $NOBITS ?AWS;
data : $NOBITS ?AW!S;
# note segment
note = NOTE;
note : $NOTE;
# non-segment
nonsegment = NONSEGMENT;
```

Default HP-UX Release 11.0 Mapfile

The HP-UX Release 11.0 64-bit linker uses the following default mapfile:

```
# text segment
text = LOAD ?RXc V0x4000000000001000;
text : .dynamic;
text : .dynsym;
text : .dynstr;
text : .hash;
text : $REL ?A;
text : $RELA ?A;
text : $UNWIND ?A;
text : $PROGBITS ?A!X!W;
text : .PARISC.milli;
text : .text;
text : $PROGBITS ?AX!W;
# data segment
data : .hdata;
data =LOAD ?RWmo V0x8000000000001000;
data : .data;
data : $PROGBITS ?AW!S;
data : .opd;
data : .plt;
data : .dlt;
data : .sdata;
data : $PROGBITS ?AWS;
data : .sbss;
data : $NOBITS ?AWS;
data : .bss;
data : $NOBITS ?AW!S;
data : .hbss
# Thread specific storage segment
thread_specific = HP_TLS ?RW;
thread_specific : .tbss;
```

```
thread_specific : $NOBITS ?AWT;
# Note segment
note = NOTE;
note : $NOTE;
# non-segment
nonsegment = NONSEGMENT;
nonsegment : .debug_header;
nonsegment : .debug_gntt;
nonsegment : .debug_ntt;
nonsegment : .debug_slt;
nonsegment : .debug_vt;
```

Defining Syntax for Mapfile Directives

A mapfile can have zero or more mapfile directives. There are two types of mapfile directives: **segment declarations** and **section mapping directives**. The directives can span across lines and are terminated by a semicolon.

The following syntax conventions are used to describe the directives:

- `[...]*` means zero or more
- `[...]+` means one or more
- `[...]` means optional
- The *section_names* and *segment_names* are the same as a C identifier except that a period (`.`) is treated as a letter.
- A number can be hexadecimal, following the same syntax as the C language.
- The section, segment, file, and symbol names are case-sensitive.
- A string of characters following # and ending at a new-line is considered a comment.

Defining Mapfile Segment Declarations

A segment declaration can create a new segment with a set of attributes or change the attributes of an existing segment.

segment_name = {*segment_attribute_value*}* ;

The segment attributes and their valid values are as follows:

Attribute	Value
segment_type	LOAD (default), HP_TLS, NOTE, NONSEGMENT
segment_flags	?[R][W][X][s][l][m][c][k][g][o]
virtual_address	*Vnumber*
physical_address	*Pnumber*
alignment	*Anumber*

- NOTE segments cannot be assigned any segment attribute other than a *segment_type*.
- If you do not specify *virtual_address*, *physical_address* and *alignment*, the linker calculates these values as it builds the executable. If you specify both a *virtual_address* and an *alignment* for a segment, the *virtual_address* value takes priority.
- An *alignment* value greater than the file system block size (4K) also specifies the page size. In that case, the value of the alignment is also the size of the page. The operating system uses the largest page size available that is no greater than the value of the alignment when mapping a segment.
- The *segment_type* NONSEGMENT describes sections placed at the end of the executable file. The linker does not create a program header entry for this segment.

Segment Flags

Segment declarations support the following segment flags:

Flag	Action
R	Readable
W	Writable
X	Executable

The default *segment_flags* for a LOADable segment is ?RWX.

Segment declarations support the following special flags:

Flag	Action
s	Enables static branch prediction on a segment. This flag is not set by default. (Dynamic branch prediction is the default.)
l	Enables lazy swap allocation for a segment. This flag is not set by default. (The lazy swap is disabled by default.)
m	Sets the “modification” hint for a segment. When this flag is set, it indicates that the program expects to modify the pages in the segment. If not set, the program does not expect to modify any pages in the segment, even though it may have permission to do so. This flag is not set by default. (The modification hint is off by default.)
c	Sets the “code” hint for a segment. When this flag is set, it indicates that the segment mostly contains code that may be executed. When not set, it indicates that it is unlikely that the segment contains code. This flag is not set by default. (The code hint is off by default.)

Flag	Action
k	Locks a particular segment into memory when loaded. This flag is set off for all segments.
g	Groups segments together. A segment declared with g flag is grouped with a segment preceding it in the mapfile. Any number of segments can be grouped together. The grouping affects the way in which addresses are assigned to segments. The segments in one group are assigned consecutive virtual addresses.
o	Tells the linker that all the segment attributes declared for this segment can be changed or modified to achieve space and/or time optimization. When this flag is set, the linker considers all other segment attribute specifications (for this segment) as *hints* and change or modify them as it thinks fit for space and/or time optimization.

Mapfile Segment Declaration Examples

- The following example declares a segment with *segment_type* LOAD and *segment_flags* readable and executable.

  ```
  text = LOAD ?RX;
  ```

- The following example declares a LOADable segment (default) with *segment_flags* readable and writable. The *virtual_address* and *alignment* values are set to `V0x80000000` and `A0x1000` respectively.

  ```
  mydata = ?RW V0x80000000 A0x1000;
  ```

Defining Mapfile Section Mapping Directives

A section mapping directive specifies how the linker should map the input section onto output segments. This directive tells the linker what attributes of a section must be matched in order to map that section into the named segment. The set of attribute values that a section must have to map into a specific segment is called the **entrance criteria**.

segment_name : {*section_attribute_value*}* ;

The section attributes and their valid values are as follows:

Section Attribute	Value
section_name	Any valid section name
section_type	`$PROGBITS`, `$NOBITS`, `$UNWIND`, `$NOTE`, `$REL`, `$RELA`
section_flags	?[[!]A][[!]W][[!]X][[!][S]][[!][T][[!][R]

Flag	Value
`A`	Allocatable (takes up virtual memory)
`W`	Writable
`X`	Executable
`S`	Short data
`T`	`TLS` (thread local storage)
`R`	Static branch prediction

- At most one *section_type* can be specified in a mapping directive.

- If a section flag is preceded by an exclamation mark (!), it indicates that the flag should not be set to meet the entrance criteria.

  If you do not specify *section_flags*, the flag can have any value to meet the entrance criteria.

  ```
  S1 : ?XR;
  ```

  The linker maps all executable sections with static branch prediction enabled onto segment S1.

- The *section_name* attribute indicates that the linker should map the input sections with the specified name onto the named segment.

  ```
  text : .rodata;
  ```

- An input section can satisfy more than one entrance criteria.

  ```
  S1 : $PROGBITS;
  S2 : $PROGBITS;
  ```

  In this case, all sections with section type $PROGBITS are mapped onto segment S1 as the first rule takes precedence.

- An AND relationship exits between attributes specified on the same line. An OR relationship exits between attributes specified for the same segment that span more than one line.

  - Example 1:

    All sections with section_type $PROGBITS and section_flags AX (allocatable and executable) are mapped onto the text segment.

    ```
    text : $PROGBITS ?AX;
    ```

  - Example 2

    ```
    text : $PROGBITS ?AX;
    text : .rodata;
    ```

    In this case, the linker maps a section onto the text segment if:

    Its section_type is $PROGBITS and section_flags is AX.

    (or)

    Its section_name is .rodata.

Internal Map Structure

The linker use a default map structure corresponding to the default mapfile. When you use the mapfile option with the `ld` command, the linker appends the default mapfile to the end of your user-specified mapfile. (You can override the default mapfile by using the `+nodefaultmap` option.)

Placement of Segments in an Executable

As it processes each segment declaration in the mapfile, the linker compares it with the existing list of segment declarations as follows:

- If the segment does not exist already, but another with the same segment_type exists, the linker adds the segment after all of the existing segments with the same segment_type.
- If no segment with the same segment_type exists, the linker adds the new segment to the list to maintain the following order based on segment_type:
  - LOAD
  - HP_TLS
  - NOTE
  - NONSEGMENT
- If segments of same type already exists, the linker adds the new segment after the last segment with the same type.

Mapping Input Sections to Segments

As each section mapping directive in a mapfile is read in, the linker creates a new entrance criteria and appends it to the existing list of entrance criteria. It applies the entrance criteria in the order in which they are specified in the mapfile. The linker maps out the input sections in the same order as their matching entrance criteria.

Figure A-1

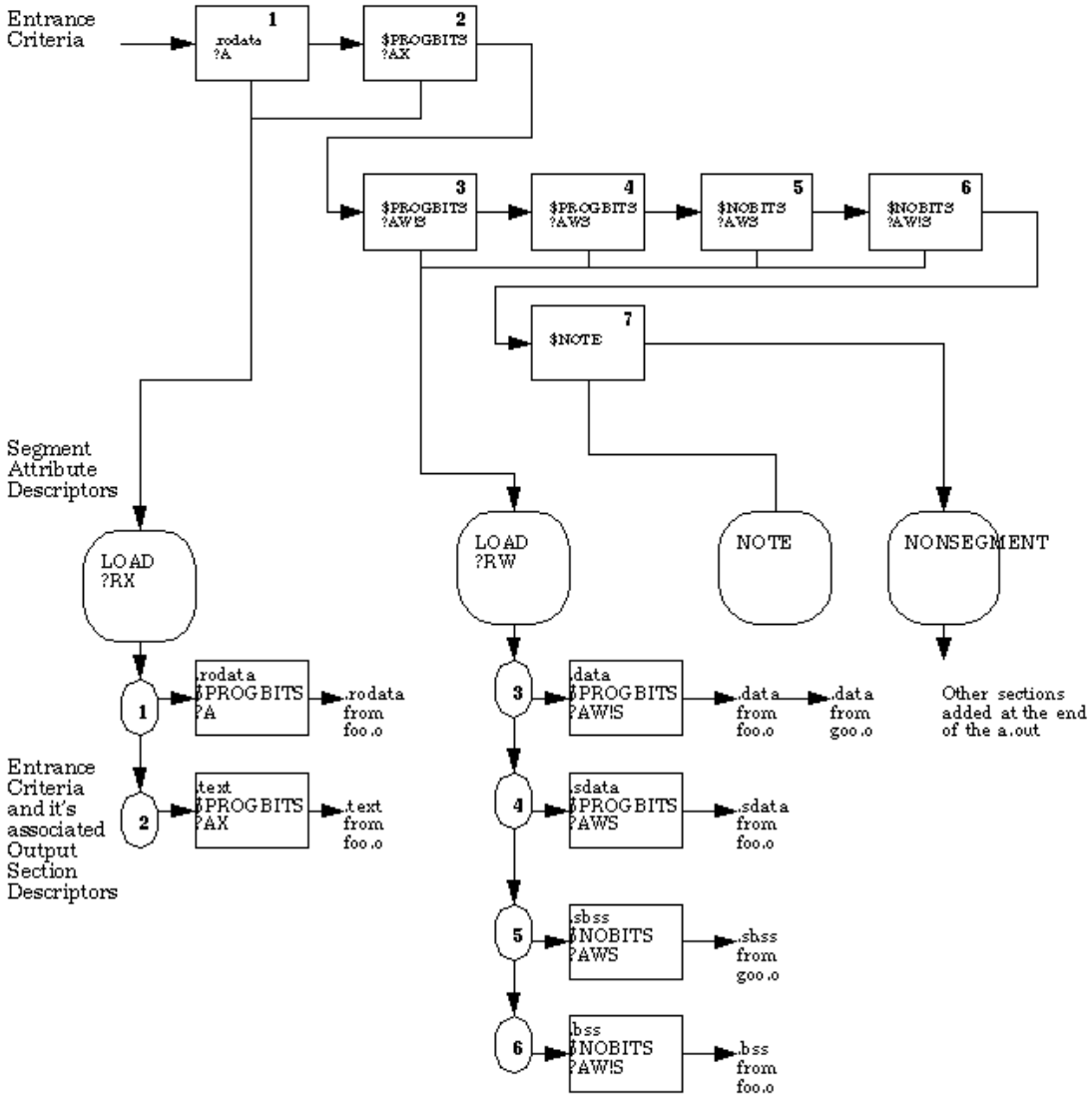


Figure A-1 shows the map structure. The *entrance criteria* boxes correspond to the information from the section mapping directives and the *segment attribute descriptors* correspond to the information from the segment declarations. The *output section descriptors* boxes group the sections that fall under each segment based on their section attributes. The linker associates each entrance criteria with a list of "output section descriptors". In Figure A-1, the entrance criteria are labeled with numbers to illustrate their associated output section descriptors.

The linker performs the following steps when mapping sections to segments:

1. When a section is read in, the liner checks the list of entrance criteria looking for a match. All specified criteria must be matched. When an entrance criteria matches, the linker traverses its associated "output section descriptor" list.
2. If the section attribute values match those of an existing output section descriptor exactly, the linker places the section at the end of the list of sections associated with that output section descriptor.
3. If no matching output section descriptor is found, but output section descriptors of the same *section_type* exists, the linker creates a new output section descriptor with the same attribute values as the section and adds that section to the new output section descriptor. It places the new output section descriptor after the last output section descriptor with the same section type.
4. If no other output section descriptor of the indicated section_type exists, the linker creates a new output section descriptor and associates the section with the new output section descriptor. It places the new output section descriptor after the last output section descriptor associated with that entrance criteria.
5. If no entrance criteria match is found, the linker places the section at the end of the "nonsegment". It does not create a program header entry for the nonsegment.

The following rules apply when the linker adds a new output section descriptor to a list of output section descriptors associated with an entrance criteria:

- If an entrance criteria selects both `$PROGBITS` and `$NOBITS` sections, the linker enforces an order such that the `$PROGBITS` sections precede `$NOBITS` sections.

- If an entrance criteria selects both S (short data) and !S (non-short data) sections, the layout of the sections depends on section_type and S flag status.The linker maintains the following order:

  $PROGBITS and !S

  $PROGBITS and S

  $NOBITS and S

  $NOBITS and !S

- The linker always tries to group all $NOBITS sections at the end of the data segment. If it does not place a $NOBITS section at the end of the data segment because of user-specified mapping directives, the linker converts that section to a $PROGBITS section and zero-fills the section contents. The linker issues a warning message when it converts a $NOBITS section into a $PROGBITS section.

Interaction between User-defined and Default Mapfile Directives

The linker adds the section mapping directives from the default mapfile after the user-specified mapping directives. The following rules apply if the you declare a built-in segment (a segment defined in the default mapfile):

- If the *segment_type* and "segment_flags" differ from the default mapfile declarations, the linker issues a warning and uses the user-specified *segment_type* and/or *segment_flags* for that segment.
- If your segment declaration does not specify a *segment_attribute_value*, the linker takes it from the default mapfile's segment declaration.
- The linker completely ignores the default mapfile if you use the option +nodefaultmap on the ld command line.

Mapfile Option Error Messages

Fatal Errors

The following conditions can result in a fatal error:

- Specifying more than one `-k` option on the command line
- Mapfile cannot be opened or read
- The linker finds a syntax error in the mapfile
- More than one segment_type, segment_flags, virtual_address, physical_address or alignment value appears on a single declaration line
- More than one section_name, section_type, or section_flags value appears on a single directive line
- A user-defined virtual address causes a segment to overlap the previous segment

Warnings

The following conditions can produce a warning message:

- A physical_address or a virtual_address value is specified for any segment other than a LOADable segment. The directive is ignored.
- A second declaration for the same segment changes an attribute value. The second declaration overrides the original.
- An attribute value for a built-in segment is changed.

Glossary

absolute object code Machine code that contains absolute virtual addresses. Created by the linker when it combines relocatable object files.

archive library A library, created by the `ar` command, which contains one or more object modules. By convention, archive library file names end with `.a`. Compare with "shared library."

attaching a shared library The process the dynamic loader goes through of mapping the shared library code and data into a process's address space, relocating any pointers in the shared library data that depend on actual virtual addresses, allocating the bss segment, and binding routines and data in the shared library to the program.

basic block A contiguous section of assembly code, produced by compilation, that has no branches in except at the top, and no branches out except at the bottom.

binding The process the dynamic loader goes through of filling in a process's procedure linkage tables and data linkage tables with the addresses of shared library routines and data. When a symbol is bound, it is accessible to the program.

breadth-first search order The dependent library search algorithm used when linking and loading 64-bit applications.

bss segment A segment of memory in which uninitialized data is stored. Compare with "text segment" and "data segment." For details, refer to *a.out*(4).

buffer A temporary holding area for data. Buffers are used to perform input and output more efficiently.

child A process that is spawned by a process (a sub-process).

code generation A phase of compilation in which object code is created.

compilation phase A particular step performed during compilation — for example, pre-processing, lexical analysis, parsing, code generation, linking.

complete executable An executable (`a.out`) file that does *not* use shared libraries. It is "complete" because all of its library

code is contained within it. Compare with "incomplete executable."

crt0.o file See **startup file**.

data export symbol An initialized global variable that may be referenced outside of the library.

data linkage table A linkage table that stores the addresses of data items.

data segment A segment of memory containing a program's initialized data. Compare with "bss segment" and "text segment." For details, refer to *a.out*(4).

deferred binding The process of waiting to bind a procedure until a program references it. Deferred binding can save program startup time. Compare with "immediate binding."

demand-loadable When a process is "demand-loadable," its pages are brought into physical memory only when they are accessed.

dependency Occurs when a shared library depends on other libraries — that is, when the shared library was built (with `ld -b...`), other libraries were specified on the command line. See also "dependent library."

dependent library A library that was specified on the command line when building a shared library (with `ld -b...`). See "dependency."

depth-first search order The dependent library search alogrithm used when linking and loading in 32-bit mode. Searching a list starting at the end of the list and moving toward the head. Shared library initialization routines are invoked by traversing the list of loaded shared libraries depth-first.

dll See "dynamic loading library."

DLT See "data linkage table."

driver A program that calls other programs.

dynamic linking The process of linking an object module with a running program and loading the module into the program's address space.

dynamic loader Code that attaches a shared library to a program. See *dld.sl*(5).

Glossary

dynamic loading library An SVR4 term for a shared library.

dynamic search path The process that allows the location of shared libraries to be specified at runtime.

entry point The location at which a program starts running after HP-UX loads it into memory. The entry point is defined by the symbol `$START$` in `crt0.o`.

explicit loading The process of using the *shl_load*(3X) function to load a shared library into a running program.

export stub A short code segment generated by the linker for a global definition in a shared library. External calls to shared library procedures go through the export stub. See also **import stub**.

export symbol A symbol definition that may be referenced outside the library.

exporting a symbol Making a symbol visible to code outside the module in which the symbol was defined. This is usually done with the `+e` or `-E` option.

external reference A reference to a symbol defined outside an object file.

feedback-directed positioning

An optimization technique wherein procedures are relocated in a program, based on profiling data obtained from running the program. Feedback-directed positioning is one of the optimizations performed during profile-based optimization.

file descriptor A file descriptor is returned by the *open*(2), *creat*(2), and *dup*(2) system calls. The file descriptor is used by other system calls (for example, *read*(2), *write*(2), and *close*(2)) to refer to a the file.

filters Programs that accept input data and modify it in some way before passing it on. For example, the `pr` command is a filter.

flush The process of emptying a buffer's contents and resetting its internal data structures.

global definition A definition of a procedure, function, or data item that can be accessed by code in another object file.

Glossary

header string A string, "!<arch>\n", which identifies a file as an archive created by `ar` (`\n` represents the newline character).

hiding a symbol Making a symbol invisible to code outside the module in which the symbol was defined. Accomplished with the `-h` linker option.

immediate binding By default, the dynamic loader attempts to bind all symbols in a shared library when a program starts up — known as "immediate binding." Compare with "deferred binding."

implicit address dependency

Writing code that relies on the linker to locate a symbol in a particular location or in a particular order in relation to other symbols.

implicit loading Occurs when the dynamic loader automatically loads any required libraries when a program starts execution. Compare with "explicit" loading.

import stub A short code segment generated by the linker for external references to shared library routines. See also **export stub**.

import symbol An external reference made from a library.

incomplete executable An executable (`a.out`) file that uses shared libraries. It is "incomplete" because it does not actually contain the shared library code that it uses; instead, the shared library code is attached when the program runs. Compare with "complete executable."

indirect addressing The process of accessing a memory location through a memory address that is stored in memory or a register.

initializer An initialization routine that is called when a shared library is loaded or unloaded.

intermediate code A representation of object code that is at a lower level than the source code, but at a higher level than the object code.

I-SOM Intermediate code-System Object Module. Used during profile-based optimizations and level 4 optimization.

library A file containing object code for subroutines and data that can be used by programs.

Glossary

link order The order in which object files and libraries are specified on the linker command line.

link-edit phase The compilation phase in which the compiler calls the linker to create an executable (`a.out`) file from object modules and libraries.

linkage table A table containing the addresses of shared library routines and data. A process calls shared library routines and accesses shared library data indirectly through the linkage table.

load graph A list of dependent shared libraries in the order in which the libraries are to be loaded by the dynamic loader. Any executable program or shared library with dependencies has a load graph.

local definition A definition of a routine or data that is accessible only within the object file in which it is defined.

lock file A file used to ensure that only one process at a time can access data in a particular file.

magic number A number that identifies how an executable file should be loaded. Possible values are `SHARE_MAGIC`, `DEMAND_MAGIC`, and `EXEC_MAGIC`. Refer to *magic*(4) for details.

man page A page in the *HP-UX Reference*. Man page references take the form *title*(*section*), where *title* is the name of the page and *section* is the section in which the page can be found. For example, *open*(2) refers to the *open*(2) page in section 2 of the *HP-UX Reference*. Or use the *man*(1) command to view man pages, for example, `man open`.

mapfile The file which describes the mapping of input sections to segments in an output file.

millicode Special-purpose routines written in assembly language and designed for performance.

nonfatal binding Like immediate binding, nonfatal immediate binding causes all required symbols to be bound at program startup. The main difference from immediate binding

is that program execution continues *even if the dynamic loader cannot resolve symbols.*

object code See **relocatable object code**.

object file A file containing machine language instructions and data in a form that the linker can use to create an executable program.

object module A file containing machine language code and data in a form that the linker can use to create an executable program or shared library.

parent process The process that spawned a particular process. See also "process ID."

PBO See "profile-based optimization."

PC-relative A form of machine-code addressing in which addresses are referenced relative to the program counter register, or PC register.

physical address A reference to an exact physical memory location (as opposed to virtual memory location).

PIC See "position-independent code."

pipe An input/output channel intended for use between two processes: One process writes into the pipe, while the other reads.

PLT See "procedure linkage table."

position-independent code

Object code that contains no absolute addresses. All addresses are specified relative to the program counter or indirectly through the linkage table. Position-independent code can be used to create shared libraries.

pragma A C directive for controlling the compilation of source.

procedure linkage table A linkage table that stores the addresses of procedures and functions.

process ID An integer that uniquely identifies a process. Sometimes referred to as "PID."

profile-based optimization A kind of optimization in which the compiler and linker work together to optimize an application based on

profile data obtained from running the application on a typical input data set.

relocatable object code

Machine code that is generated by compilers and assemblers. It is relocatable in the sense that it does not contain actual addresses; instead, it contains symbols corresponding to actual addresses. The linker decides where to place these symbols in virtual memory, and changes the symbols to absolute virtual addresses.

relocation The process of revising code and data addresses in relocatable object code. This occurs when the linker must combine object files to create an executable program. It also occurs when the dynamic loader loads a shared library into a process's address space.

restricted binding A type of binding in which the dynamic loader restricts its search for symbols to those that were visible when a library was loaded.

RPATH The variable which contains the search path for dynamic libraries.

section mapping directive A mapfile directive which specifies how the linker should map the input sections onto the output segments.

segment declaration A mapfiel directive which creates a new section or edits the attributes of an existing segment.

shared executable An `a.out` file whose text segment is shareable by multiple processes.

shared library A library, created by the `ld` command, which contains one or more PIC object modules. Shared library file names end with `.sl`. Compare with "archive library."

shared library handle A descriptor of type `shl_t` (type defined in `<dl.h>`), which shared library management routines use to refer to a loaded shared library.

standard error The default stream for sending error messages — usually connected to the screen.

standard input The default stream for collecting character input data — usually connected to the keyboard.

standard input/output library

A collection of routines that provide efficient and portable input/output services for most C programs.

standard output The default stream for sending character output data — usually connected to the screen.

startup file Also known as `crt0.o`, this is the first object file that is linked with an executable program. It contains the program's entry point. The startup code does such things as retrieving command line arguments into the program at run time, and activating the dynamic loader (*dld.sl*(5)) to load any required shared libraries.

storage export symbol An uninitialized global variable that may be referenced outside of the library.

stream A data structure of type `FILE *` used by various input/output routines.

stub A short code segment inserted into procedure calling sequences by the linker. Stubs are used for very specific purposes, such as inter-space calls (for example, shared-library calls), long branches, and preserving calling interfaces across modules (for example, parameter relocation). Refer to the manual *PA-RISC Procedure Calling Conventions Reference Manual*. See also **import stub** and **export stub**.

supporting library A library that was specified on the command line when building a shared library (with `ld -b`...). Same as **dependent library**.

symbol name The name by which a procedure, function, or data item is referred to in an object module.

symbol table A table, found in object and archive files, which lists the symbols (procedures or data) defined and referenced in the file. For symbols defined in the file, an offset is stored.

system calls System library routines that provide low-level system services; they are documented in section 2 of the *HP-UX Reference*.

text segment A segment of read-only memory in which a program's machine language instructions are

typically stored. Compare with "bss segment" and "data segment." For details, refer to *a.out*(4).

umask A field of bits (set by the *umask*(1) command) that turns off certain file permissions for newly created files.

version number A number that differentiates different versions of routines in a shared library.

wrapper library A library that contains alternate versions of library functions, each of which performs some bookkeeping and then calls the actual function.

Glossary

Index

Index

Index

Index

Index

Index

Index

Index

Index