HEWLETT® PACKARD

HP 9000 Computers

Programming on HP-UX

Programming on HP-UX

HP 9000 Computers

HP Part No. B2355-90026
Printed in USA August 1992

E0892

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.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Trademarks. The following trademark is used 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.

August 1992 Edition 1. This manual supersedes *Programming on HP-UX* (part number B2355-90010). The main reason for this new edition is to document new functionality for the HP-UX 9.0 release:

- Series 300/400/800 computers now support the following features, which formerly were supported only on Series 700 systems:
  - `shl_gethandle` routine
  - `-h` and `+e` linker options
  - `BIND_VERBOSE`, `BIND_NONFATAL`, and `BIND_FIRST` flags to the `shl_load` routine
  - `-B nonfatal` linker option
  - `-c` linker option (for linker option files)
- Linker functionality is now identical on Series 800 and Series 700.
- A shared library's run-time location can be different than its link-time location; in other words, shared libraries no longer have to reside at the same location at run time as they were when the application was linked.
- Shared libraries can now be debugged with `xdb`.
- There is a new section on improving the run-time performance of shared libraries.
- Profile-based optimization (PBO) has changed somewhat.

- On Series 700/800, greater math library performance can be obtained by linking with the PA89 math libraries (as described in Chapter 2).
- On Series 700/800, there is a new shared library management routine, `shl_getsymbols`.
- New `BIND_RESTRICTED` flag to the `shl_load` routine.
- New `-B restricted` linker option.
- New `+I` linker option (shared library initializers).

The previous edition (part number B2355-90010) superseded part number B1864-90007. The main reason for the previous edition was to include information from the *Programming on HP-UX Technical Addendum* for the Series 700 HP-UX release 8.05 (part number B2355-90604):

- profile-based optimization—repositioning procedures in an `a.out` file to optimize run-time performance
- linker optimization with the `-O` option
- new linker option for shared library binding: `-B nonfatal`
- explicitly hiding and exporting shared library symbols with the `-h` and `+e` linker options
- support for linker option files with the `-c` option
- shared library dependencies—automatically loading libraries that are required by other libraries
- new `shl_load` flags—`BIND_VERBOSE`, `BIND_FIRST`, and `BIND_NONFATAL`
- new shared library management routines—`shl_definesym` and `shl_gethandle`
- support for shared library debugging with the `-s` option to `xdb`

Contents

Figures

Tables

1

Introduction

This chapter describes

- the scope of this manual
- what you should know before reading this manual
- what manuals to go to for additional information
- the content of each chapter
- conventions used throughout this manual

Manual Contents

This book describes the fundamentals of software development on HP-UX. It shows how the basic pieces of the HP-UX software development environment fit together—the compilers, assemblers, linker, libraries, and object files. It also describes

- the two kinds of subroutine libraries (archive and shared), how to create them, and how to link them with your programs
- using the linker, `ld`, to create executable programs
- special considerations for writing position-independent code, which is the code used to build shared libraries
- managing shared libraries from within a program
- using the standard I/O library
- using character, string, and date/time manipulation routines from the standard library
- using math routines found in the standard C library (`libc`) and the math library (`libm`)
- advanced system programming techniques
- general-purpose software development tools:
  - `make`—a program for maintaining computer programs.
  - `m4`—the macro preprocessor.
  - SCCS—a source code control systems (SCCS).

This book does *not* discuss in detail the compilers (`cc`, `f77`, `pc`), debuggers, or language-specific tools (such as `cflow`, `ratfor`, and `lint`). For details on where to look for a specific topic not covered in this manual, see "Related Manuals".

Prerequisites

Before reading this manual, you should have a good grasp of these basic HP-UX concepts:

- login and logout
- shells
- environment variables: `PATH`, `HOME`, `TERM`
- standard input, standard output, and standard error output
- processes
- input/output redirection
- pipes
- text editing (for example, with the `vi` text editor)

For details on these and other important prerequisite concepts, refer to

- *A Beginner's Guide to HP-UX*
- *How HP-UX Works: Concepts for the System Administrator*

Related Manuals

Table 1-1. Related Manuals

For Information On ...	See This Manual ...
Floating-point programming (Series 700/800 computers)	*HP-UX Floating-Point Guide*
`xdb` debugger	*HP-UX Symbolic Debugger User's Guide*
Porting programs across different HP computer systems and from other vendors' systems to HP-UX	*HP-UX Portability Guide*
Assembly language programming	Series 700/800 computers: ■ *Assembly Language Reference Manual* Series 300/400 computers: ■ *HP-UX Assembler and Tools*
C Programming	Series 700/800 computers: ■ *HP C/HP-UX Reference Manual* ■ *HP C Programmer's Guide* ■ *C Programming Tools* Series 300/400 computers: ■ *C: A Reference Manual* (2nd Edition; Harbison & Steele) ■ *C Programmer's Guide* ■ *C Programming Tools*
C++ Programming	■ *HP C++ Programmer's Guide* ■ *C++ Quick Reference Card* ■ *The C++ Programming Language* (2nd Edition; Stroustrup)

Table 1-1. Related Manuals (continued)

For Information On ...	See This Manual ...
FORTRAN Programming	■ *FORTRAN/9000 Programmer's Reference* ■ *FORTRAN/9000 Programmer's Guide*
Pascal Programming	Series 700/800 computers: ■ *HP Pascal/HP-UX Reference Manual* ■ *HP Pascal Programmer's Guide* Series 300/400 computers: ■ *Pascal Language Reference*
COBOL Programming	■ *COBOL/HP-UX Implementation Notes* ■ *COBOL/HP-UX Language Reference Manual* ■ *COBOL/HP-UX Operating Manual* ■ *COBOL/HP-UX Utilities Manual* ■ *COBOL/HP-UX Pocket Guide*
RCS	*HP-UX Reference*: *rcs*(1), *co*(1), *ci*(1), *rcsdiff*(1), *rcsintro*(5), *rlog*(1), *rcsfile*(4), *acl*(5).

Chapter Summaries

Table 1-2. Summary of Chapter Contents

Chap	Title Description
1	*Introduction* What's in this manual.
2	*The HP-UX Software Development Environment* Describes the fundamentals of developing programs on HP-UX. Introduces HP-UX compilers, object files, libraries (archive and shared), the linker (`ld`), `a.out` files, the assemblers, and other useful programming tools. *You should understand the concepts in this chapter before proceeding with other chapters.*
3	*Creating Archive Libraries* Creating archive libraries using the **ar** command.
4	*Creating Shared Libraries* Creating shared libraries using the `ld` command on object files containing position-independent code (PIC).
5	*Linking and Running Programs* Using the linker, `ld`, to create executable programs; linking with archive and shared libraries; migrating to shared libraries from archive libraries; differences at run time between archive and shared libraries.
6	*Profile-Based Optimization and Linker Optimization* How to use profile-based optimization to improve run-time performance, and how to optimize access to data with linker optimization.
7	*Position-Independent Code* Relocatable object code, position-independent code (PIC), and how the compilers generate certain language constructs in PIC. Describes PIC for Series 300/400 and Series 700/800.
8	*Shared Library Management Routines* How to explicitly load libraries at run time using shared library management routines. This is useful mainly when it is impossible to know the name of libraries at link time.

Table 1-2. Summary of Chapter Contents (continued)

Chap	Title Description
9	*Standard Input/Output Library Routines* Using standard library (`libc`) input/output routines to read/write from/to the keyboard/screen, files, or strings.
10	*Standard Character, String, and Date Manipulation Routines* Describes standard library (`libc`) routines that manipulate characters and strings.
11	*Standard Math Routines* Describes math routines from the standard C library (`libc`) and standard math library (`libm`).
12	*Advanced HP-UX Programming* Describes how to access command line arguments from C programs, how to handle errors, and how to use system calls, which provide low-level access to the kernel for input/output, process control, and signal handling.
13	*make: A Program for Maintaining Computer Programs* Describes how to use the **make** program for managing compiles.
14	*SCCS: Source Code Control System* Describes the use of the SCCS source code control system.
15	*The m4 Macro Preprocessor* Describes how to use the **m4** macro preprocessor.
Glossary	*Glossary* Contains definitions of important terms used throughout this manual.

Conventions

Table 1-3 summarizes the typographical conventions used throughout this manual.

Table 1-3. Typographical Conventions

Convention	Description
`computer font`	Denotes information displayed by the computer (for example, `login:`), file names (for example, `/usr/include/stdio.h`), and command names (for example, `vi`).
underlining	Denotes text you must type explicitly: `$ cc -c prog.c -lm`
name(N)	Refers to a command, system call, or library routine in the *HP-UX Reference*. N refers to the section in which *name* can be found. For example, *ld*(1) refers to the *ld* page in section 1 of the *HP-UX Reference*.
italic	Denotes information that you must fill in—for example: `cc -o` *outfile progfile*`.c` means that you should specify your own *outfile* and *progfile* name.

Most of the programming examples presented in this manual are in the ANSI C language. When compiling ANSI C examples, be sure to specify the `-Aa` compiler option.

Although most of the examples are in C, the concepts presented apply equally well in most cases to other HP-UX languages, especially FORTRAN, C++.

2

The HP-UX Software Development Environment

Because HP-UX has such a powerful, versatile programming environment, there are many different ways to do things, and no one way is absolutely right. Rather than trying to describe every possible way, this chapter introduces program development with a simple example in the first section. Subsequent sections build on the commands and concepts introduced in the first section. Specific topics introduced in this chapter are:

- compilers
- object (`.o`) files
- a command for viewing symbols in object files (`nm`)
- the linker (`ld`)
- executable (`a.out`) files
- assemblers (`as`)
- libraries—archive (`.a`) and shared (`.sl`)
- system libraries (`libc`, `libm`, etc)
- other programming tools:
  - debuggers (`xdb` and `adb`)
  - source code control systems (RCS and SCCS)
  - program compilation manager (`make`)
- the *SoftBench*™ development environment

After finishing this chapter, you should be able to develop your own programs using HP-UX compilers and the tools described in the rest of this manual.

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 in Figure 2-1, 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
    :          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 */
}
```

Figure 2-1. sumnum.c—Sum the Numbers from 1 to n

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 Figure 2-2.

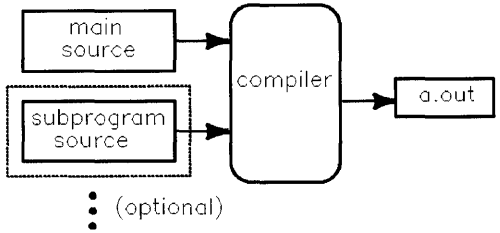


Figure 2-2. High-Level View of the Compiler

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.
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.

Figure 2-3 summarizes how a compiler driver works.

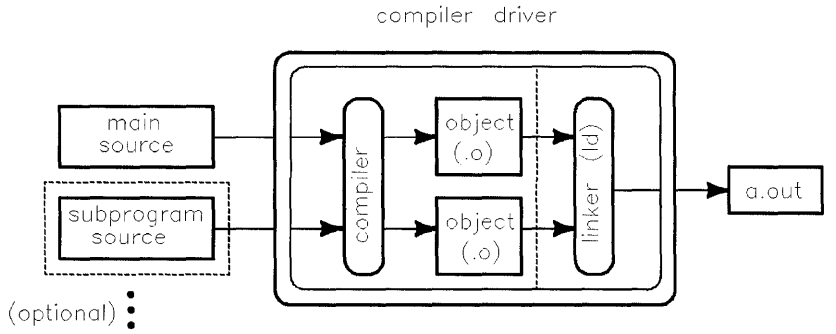


Figure 2-3. Looking "inside" a Compiler

The 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
cc: CCOPTS is not set.
main.c:
/lib/cpp.ansi main.c /tmp/ctmAAAa09888 -D__hp9000s700 \
  -D__hp9000s800 -D__hppa -D__hpux -D__unix -D_PA_RISC1_1 \
  -A -I /usr/include
cc: Entering Preprocessor.
/lib/ccom /tmp/ctmAAAa09888 main.o -O0 -v -Aa
```

```
func.c:
/lib/cpp.ansi func.c /tmp/ctmAAAa09888 -D__hp9000s700 \
  -D__hp9000s800 -D__hppa -D__hpux -D__unix -D_PA_RISC1_1 \
  -A -I /usr/include
cc: Entering Preprocessor.
/lib/ccom /tmp/ctmAAAa09888 func.o -O0 -v -Aa
cc: LPATH is /lib/pa1.1:/usr/lib/pa1.1:/lib:/usr/lib
/bin/ld /lib/crt0.o -u main main.o func.o -lc
cc: Entering Link editor.
```

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

Compiling `main.c` and `func.c` with `-v` on a Series 300/400 computer produced this output:

```
$ cc -v -Aa main.c func.c
main.c:
   /lib/ccom.ansi:  cpp.ansi main.c -I/usr/include -D__hp9000s300 \
      -D__unix-D__hpux ccom.ansi -YS |
   /bin/as:  as -o main.o
func.c:
   /lib/ccom.ansi:  cpp.ansi func.c -I/usr/include -D__hp9000s300 \
      -D__unix-D__hpux ccom.ansi -YS |
   /bin/as:  as -o func.o
/bin/ld  /lib/crt0.o main.o func.o -x -lc
```

The compiler creates an object file for each source file specified on the command line. The files are placed in the current working directory. Each object file has the same name as its corresponding source file, except that the language suffix (e.g., `.c`, `.f`, `.p`) is replaced with `.o`. For instance, in the above example, two object files were created, `main.o` and `func.o`, which we can see with the `ls` command:

```
$ ls *.o                    List all .o files.
func.o      main.o          Here are the object files.
```

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.

Using nm to View Symbols

To view the symbols defined in an object file, use the **nm** command. Its syntax and output differ slightly on Series 300/400 and Series 700/800 computers, but it provides basically the same information on both systems. On Series 700/800 computers, **nm** produces output similar to Series 300/400 computers if invoked with the **-p** option. Figure 2-4 shows output from running **nm -p** on the **func.o** and **main.o** object files on a Series 700/800 computer; Figure 2-5 shows the output produced on a Series 300/400 computer.

```
$ nm -p func.o
1073741824 d  $THIS_DATA$          Other symbols created from compiling.
1073741824 d  $THIS_SHORTDATA$
1073741824 b  $THIS_BSS$
1073741824 d  $THIS_SHORTBSS$
0000000000 T  sum_n                Global definition 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               External reference to printf.
0000000000 U  scanf                External reference to scanf.
0000000000 U  sum_n                External reference to sum_n.
```

Figure 2-4. Output of nm on a Series 700/800 Computer

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 bss (uninitialized data area).

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).

```
$ nm func.o                    View symbols in func.o.
0x00000000 T  _sum_n           The global definition for sum_n appears.
$ nm main.o                    View symbols in main.o.
0x00000000 T  _main            Global definition for main.
0x00000000 U  _printf          External reference to printf.
0x00000000 U  _scanf           External reference to printf.
0x00000000 U  _sum_n           External reference to sum_n.
```

Figure 2-5. Output of nm on a Series 300/400 Computer

Notice that on Series 300/400 computers, global symbols all begin with an underscore (for example, `_sum_n`); on Series 700/800 computers, they do not.

The Link-Edit Phase of Compilation

When called in the link-edit phase of compilation, `ld` builds an `a.out` file from the object files passed by the compiler. `ld` attempts to match external references with global definitions. For instance, in the C program example (see Figure 2-4), `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 2-6).

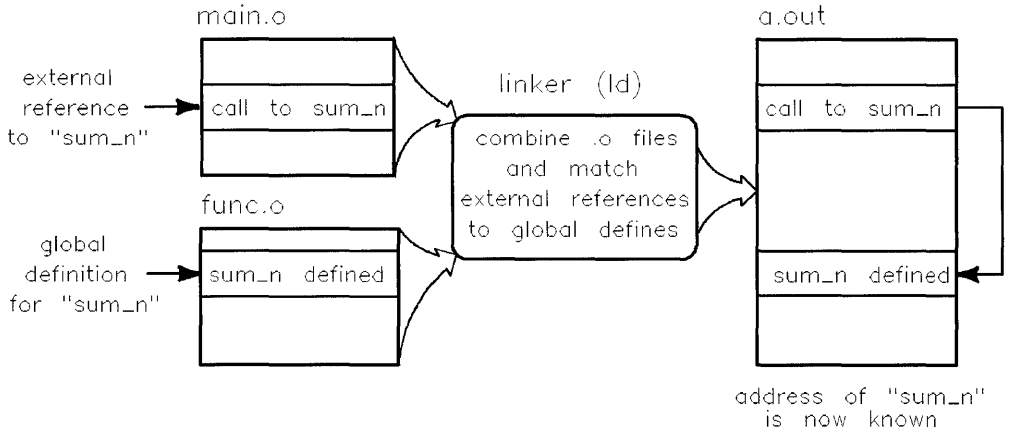


Figure 2-6. Matching the External Reference to sum_n

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 on a Series 700/800 computer:

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

Similar output is produced on Series 300/400 computers:

```
$ cc main.c
ld: Undefined external -
        _sum_n
ld: output file still contains undefined symbols
```

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, FORTRAN, and Pascal programs to do input, output, and other standard operations.

Library Naming Conventions

By convention, library names have the form:

`lib`*name*`.`*sfx*

name is a string of one or more characters that identifies the library.

sfx is `.a` if the library is an archive library or `.sl` if the library is a shared library. For details on archive versus shared libraries, see the section "Archive and Shared Libraries" later in this chapter.

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

Specifying Libraries to the Linker (-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`.

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 in this Series 700/800 example:

```
$ cc -v main.c func.c                          Compile with -v to see
    :                                          the ld command line.
/bin/ld /lib/crt0.o -u main main.o func.o -lc  Notice -lc at end.
cc: Entering Link editor.
```

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

```
$ f77 -v sumnum.f
   :
/bin/ld -x /lib/crt0.o sumnum.o -lcl -lisamstub -lc
```

Specifying Libraries on the Compile Line (-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 -v cursesprog.c -lcurses
   :
/bin/ld /lib/crt0.o -u main main.o -lcurses -lc
cc: Entering Link editor.
```

Linking with the crt0.o Startup File

Notice also, in the above example, that the compiler linked `cursesprog.o` with the file `/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 Chapter 5.

The Default Library Search Path

By default, `ld` searches for libraries in the `/lib` and `/usr/lib` directories, in that order. (If the `-p` or `-G` profiling option is specified on the command line, the compiler directs the linker to also search `/lib/libp`.) The default order can be overridden with the `LPATH` environment variable or the `-L` linker option. `LPATH` and `-L` are described in detail in Chapter 5. The `-L` option is discussed later in the section "Compiler Options That Affect the Linker".

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.

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.

Table 2-1 summarizes the major library groups defined in the *HP-UX Reference*.

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`, etc.) are documented in the FORTRAN language documentation.

Table 2-1. Libraries Documented in the HP-UX Reference

Group	Description
(2)[1]	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)[1]	These are standard *C* library routines located in `libc`. These routines are described in Chapter 9 through Chapter 12.
(3S)[1]	These functions comprise the *S*tandard input/output routines (see *stdio*(3S)). They are located in `libc`. These routines are described also in Chapter 9.
(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.

1 The routines marked by (2), (3C), and (3S) comprise the standard C library `libc`. The C, FORTRAN, and Pascal compilers automatically link with this library when creating an executable program.

Archive and Shared Libraries

HP-UX supports two kinds of libraries: archive and shared. Archive libraries are the more traditional of the two.

Almost all system libraries are available as archive. In most cases, a shared version is also available. Archive library file names end with `.a`; shared library file names end with `.sl`. If both versions of a library exist, they are usually found in the same directory. For example, the archive `libc` is named `/lib/libc.a`; the shared version is named `/lib/libc.sl`.

If both 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:

```
ld /lib/crt0.o prog.o -lc
```

This instructs the linker to search the C library, `libc`, to resolve unsatisfied references from `prog.o`. If a shared `libc` exists (`/lib/libc.sl`), `ld` uses it instead of the archive `libc` (`/lib/libc.a`). You can, however, override this behavior, and select the archive version of a library (see Chapter 5).

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 Chapter 3. To create shared libraries, use `ld` to combine object files containing position-independent code (PIC), as described in Chapter 4.

Table 2-2 summarizes differences between archive and shared libraries. A detailed discussion of archive and shared libraries follows the table.

Table 2-2. Comparison of Archive and Shared Libraries

Comparing	Archive	Shared
file name suffix	Suffix is `.a`.	Suffix is `.sl`.
object code	Made from 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).
creation	Combine object files with the **`ar`** command (see Chapter 3).	Combine PIC object files with the **`ld`** command (see Chapter 4).
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. Does, however, contain some 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.

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.

For example, suppose you write a C program that calls `printf` from the `libc` library. Figure 2-7 shows how the resulting `a.out` file would look if you linked the program with the archive version of `libc`.

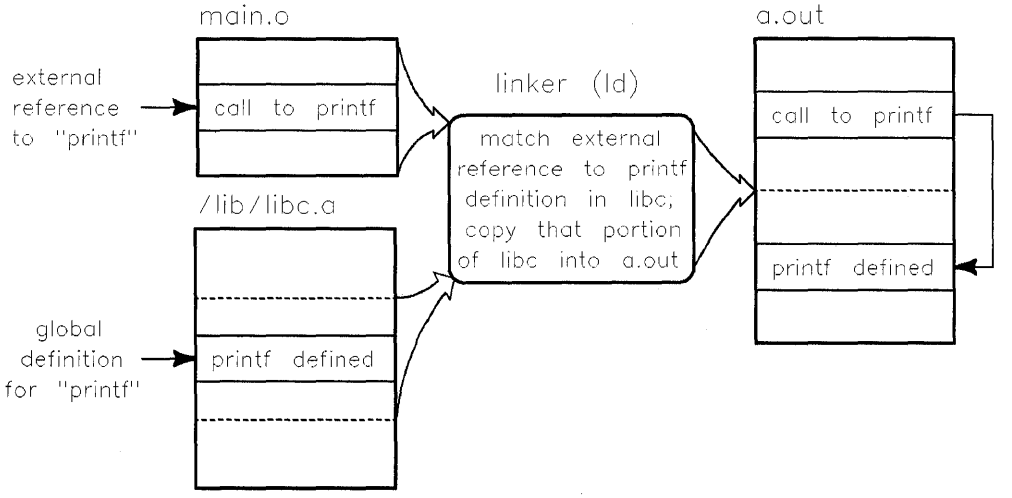


Figure 2-7. Linking with an Archive Library

What Are Shared Libraries?

Like an archive library, a **shared library** contains relocatable object code. However, `ld` treats shared libraries quite differently than 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.

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. 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.

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 2-8 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).

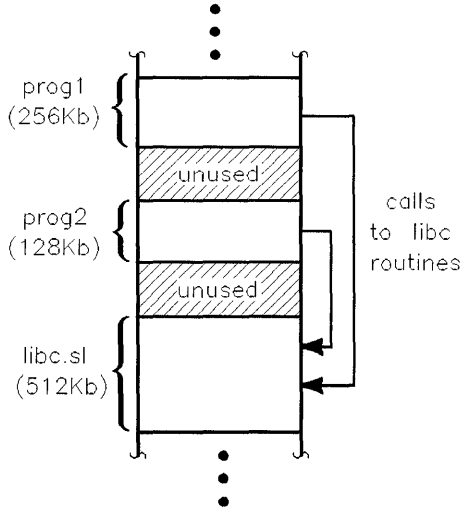


Figure 2-8. Two Processes Sharing libc

Compare this with the memory requirements if `prog1` and `prog2` had been linked with the archive version of `libc`. As shown in Figure 2-9, 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.

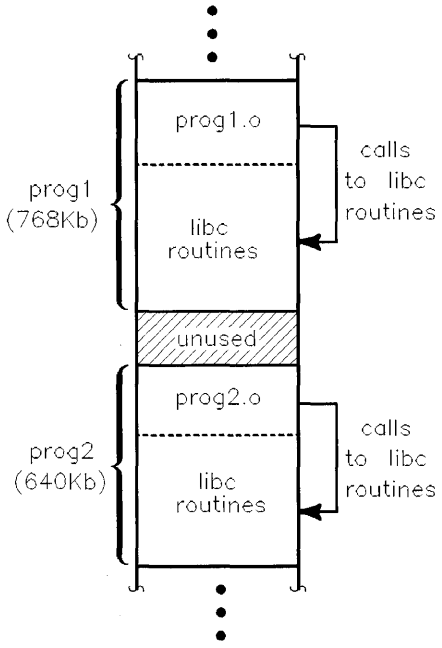


Figure 2-9. Two Processes with Their Own Copies of libc

Position-Independent Code

Shared libraries are constructed from object files that contain a special kind of object code known as **position-independent code** (**PIC**). All that most users need to know about PIC is that it has characteristics that make it shareable by multiple processes, and you create it by compiling with the `+z/+Z` compiler option (see Chapter 4).

If you really need to know why: PIC makes sharing possible because it contains no absolute virtual addresses; only PC-relative addressing is used. (**PC-relative** addressing means that all addresses are referenced relative to the program counter register.) Therefore, PIC can be placed anywhere in a process's address space without addresses having to be relocated. For details on position-independent code, see Chapter 7.

Compiler Options That Affect the Linker

This section summarizes compiler options—common to the C, FORTRAN, and Pascal compilers—that control how the compiler interacts with the linker. (For more information on linker options, see Chapter 4 and Chapter 5.)

Renaming the a.out File (-o name)

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
```

Suppressing the Link-Edit Phase (-c)

The `-c` 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, which can be linked later with other object files. 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.
```

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.

Getting Verbose Output (-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 Series 700/800 Pascal compiler shows that it automatically links with `libcl`, `libm`, and `libc`.

```
$ pc -v prog.p
/usr/lib/pascomp prog.p ? prog.o . ?  ?
/bin/ld /lib/crt0.o prog.o -lcl -lm -lc -z
unlink prog.o
```

Using the `-v` option with the Series 300/400 Pascal compiler shows that it automatically links with `libpc`, `libm`, and `libc`:

```
$ pc -v prog.p
    :
pc: /bin/ld /lib/crt0.o prog.o -x -lpc -lm -lc
```

Passing Linker Options Directly (-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. The command for telling the linker to use an archive version of `libm` is:

```
$ ld /lib/crt0.o mathprog.o -a archive -lm -a shared -lc
```

To pass `-a archive` directly to the linker from the C command line, use `-Wl` as follows:

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

Augmenting the Default Linker Search Path (-Wl,-L)

By default, the linker searches the `/lib` and `/usr/lib` directories for libraries specified with the `-l` 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`) and the POSIX FORTRAN compiler (`fort77`) 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` via the `-Wl` option. 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
```

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 "Specifying Libraries (-l)" in Chapter 5.

Selecting Faster Libraries (Series 700/800 Only)

On Series 700/800, some libraries—for example, the math libraries `libm` and `libM`—are provided in two versions: PA-RISC 1.0 (PA1.0) and PA-RISC 1.1 (PA1.1). Derived from an earlier PA-RISC instruction set, PA1.0 libraries are completely compatible between Series 700 and 800. PA1.1 libraries, on the other hand, take advantage of the latest improvements in the PA-RISC instruction set, resulting in faster code. However, PA1.1 libraries run only on Series 700 models and Series 800 models whose last digit is 7 (that is, $8x7$ models). (The file `/usr/lib/sched.models` shows which architecture is used for a particular model.)

There are primarily two types of application developers who will need PA1.0 libraries:

- developers who have to create programs that will run on older Series 800 models (that is, those whose model numbers do *not* end with 7)
- developers who require stability and reproducibility of results more than higher performance and greater precision

The PA1.0 libraries are stored in the usual system library directories, `/lib` and `/usr/lib`. The corresponding PA1.1 libraries (if they exist) are stored in the directories `/lib/pa1.1` and `/usr/lib/pa1.1`, respectively. For example, the PA1.0 archive library `libm` is `/lib/libm.a`, while the PA1.1 version is `/lib/pa1.1/libm.a`. Note that there are no shared versions of `libm` or `libM`.

(For details on PA1.0 and PA1.1 math libraries, refer to the *HP-UX Floating-Point Guide*.)

From the Linker Command Line

To link against PA1.1 libraries from the linker command line, use the linker options `-L/lib/pa1.1` and `-L/usr/lib/pa1.1`. The `-L`*path* option causes the linker to look in the specified *path* for libraries before looking in the usual places (`/lib` and `/usr/lib`). Thus, if the PA1.1 version of a library exists, the linker will find it and use it before the PA1.0 version. For example, the following linker command will link against the PA1.1 `libc` and `libm`:

```
$ ld /lib/crt0.o -u main -L/lib/pa1.1 prog.o -lc -lm
```

Notice that `-L/usr/lib/pa1.1` was not specified in the above example. This is because the PA1.0 versions of both libraries are found in `/lib`; therefore, it is really only necessary to search `/lib/pa1.1`. If, however, the program were linked with a PA1.1 library found in `/usr/lib`, you *would* specify `-L/usr/lib/pa1.1`.

From the Compiler Command Line

To select PA1.0 or PA1.1 libraries from the compile line, use the `+DA`*arch* option, where *arch* is `1.0` for PA1.0 and `1.1` for PA1.1. Not only does `+DA` cause the compiler to invoke the linker with the correct search path, it also causes the compiler to generate PA1.0 or PA1.1 code for each object file specified on the command line.

On Series 800 computers, the default value for the `+DA` option is `+DA1.0` because the primary concern for most Series 800 applications is compatibility across all Series 700/800 models. On Series 700 computers, the default value for the `+DA` option is `+DA1.1` because the primary concern for most Series 700 applications is maximum performance. For details on the `+DA` option, refer to your language reference manual.

Restrictions on Using Faster Libraries

Here are some restrictions on using PA1.1 libraries:

- Applications built using PA1.0 libraries run on both Series 800 and 700. However, applications built with PA1.1 libraries run only on Series 700 models and Series 800 models whose last digit is 7.
- The PA1.1 versions of the math libraries are available only on systems running HP-UX release 9.0 or later.

The Assemblers

In addition to the standard programming languages, HP computers support assembly language. Although Series 300/400 and Series 700/800 computers have different architectures, there are some similarities in the assembler on both computers.

On both systems, the assembler is invoked with the `as` command. Like a compiler, an assembler reads a program (in assembly language) and produces a corresponding object file. Unlike a compiler, however, it does not call the linker. Assembly language file names end with `.s`.

Interestingly, Series 300/400 C and FORTRAN compilers convert source programs to assembly language as an intermediate phase. They then run the intermediate assembly language through `as` to produce the `.o` file. To suppress the assembly phase on Series 300/400, invoke the C or FORTRAN compiler with the `-S` option, which produces `.s` files instead of `.o` files. For example, compiling a C program with the `-S` option produced the assembly language file shown below:

```
$ cc -Aa -S hello.c          Suppress assembler phase with -S.
$ cat hello.s                View the assembly language.
        global  _main
_main:
        link.l  %a6,&LF1
        movm.l  &LS1,(%sp)
       :
```

Series 700/800 compilers do *not* run the assembler as an intermediate phase, but they still produce assembly language output if invoked with the `-S` option. For example, compiling a FORTRAN program with `-S` produced the assembly language output shown below:

```
$ f77 -S prog.f
$ more prog.s

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=44,CODE_ONLY
stuff
_start
        .PROC
        .CALLINFO CALLER,FRAME=0,SAVE_SP,SAVE_RP
        .ENTRY
        STW     2,-20(0,30)     ;offset 0x0
        LDO     48(30),30       ;offset 0x4
          :
```

Also, both assemblers support instructions and pseudo-ops for generating PIC, used to create shared libraries. Writing assembly code that produces PIC object code is described in Chapter 7 and in the following assembly language manuals:

- *Assembly Language Reference Manual* (Series 700/800).
- *HP-UX Assembler and Tools* (Series 300/400)

Other Programming Tools

In addition to the programming tools discussed thus far, HP-UX provides a rich environment of programming tools, summarized in Table 2-3.

Table 2-3. Programming Environment Tools

Tools	Description
debuggers: `xdb` and `adb`	Help you find run-time errors in programs. (See *The HP-UX Symbolic Debugger User's Guide*.)
profilers: `prof`, `gprof`	Help you locate parts of a program most frequently executed (that is, possible bottlenecks); using this data, you may be able to improve a program's performance. (See *prof*(1) and *gprof*(1).)
SCCS RCS	Source code control systems, which help manage software projects with multiple programmers. (See Chapter 14 and *rcs*(1).)
`chatr`	Changes an `a.out` file's internal attributes. (See Chapter 5 and *chatr*(1).)
`file`	Determines a file's type and lists its attributes. (See *file*(1).)
`lorder`	Determines object file dependencies; used with `tsort` command to generate more efficient link order for `ld` on Series 300/400. (See Chapter 4 and *lorder*(1).)
`m4`	A macro preprocessor, which can be used by all languages. (See Chapter 15.)
`make`	A tool for managing program "builds," compilation, and linking. (See Chapter 13.)
`nm`	Displays symbol table information in object files. (See "What Is an Object File?" in this chapter and *nm*(1).)
`od`	Shows octal or hexadecimal dumps of binary files. (See *od*(1).)
`strings`	Displays all the printable strings in an object or other binary file. Useful for seeing the strings in an `a.out`. (See *strings*(1).)
`strip`	Strips symbol table and line number information from an object file, thus making it smaller, but unusable by symbolic debuggers. Useful after a program is debugged. (See *strip*(1) and the description of the `-s` option in *ld*(1).)

SoftBench

SoftBench is an integrated set of window-based programming tools and an framework for integrating other tools. Together they provide a development environment targeted at the program construction, test, and maintenance phases of software development.

The Programming Tools

There are several programming tools in SoftBench:

- Program Editor and Program Builder address the program construction phase, and are used to develop an executable program.
- A symbolic Program Debugger and Static Analyzer are used for program analysis. Program Debugger is mainly used during the testing phase, helping to identify bugs in the executable program. Static Analyzer is most valuable in the maintenance phase, providing information on program structure to engineers who fix bugs and enhance existing programs.
- A Development Manager is used to manage the files over which the other tools operate. In particular, it organizes and maintains the program's source files during the development process.

The SoftBench Framework

The SoftBench Framework provides the environment with

- a multi-window, graphical user interface that is common throughout the environment
- a pervasive, interactive help system
- communication between the tools, allowing them to cooperate to accomplish tasks
- support for both distributed and local tool execution and data accessing

SoftBench Encapsulator

SoftBench Encapsulator delivers the customizability benefit of SoftBench to the customer. It allows customers to customize and extend the SoftBench environment by

- automating custom development processes (SoftBench Encapsulator is used to define actions that will be executed whenever specific events occur in the SoftBench environment.)
- adding the SoftBench graphical user interface to existing UNIX utilities and customer tools, without modifying the source code (The tools must use standard input and standard output.)
- adding the SoftBench graphical user interface and inter-tool messaging to C or C++ programs with simple library calls for SoftBench Encapsulator functions

3

Creating Archive Libraries

As discussed in Chapter 2, HP-UX provides many useful libraries of routines you can call from your programs. You can also create your *own* libraries. There are two kinds of libraries to create—archive and shared. This chapter describes how to create your own *archive* libraries with the `ar` command. Specifically, it discusses

- creating an archive library
- viewing an archive library's contents
- replacing object modules in an archive library
- adding object modules to an archive library
- deleting object modules from an archive library
- summary of `ar` command keys
- where to put archive libraries

For details on creating *shared* libraries, see Chapter 4. For details on linking archive libraries with programs, and for a summary of the tradeoffs between using archive versus shared libraries, see Chapter 5.

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 3-1 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.

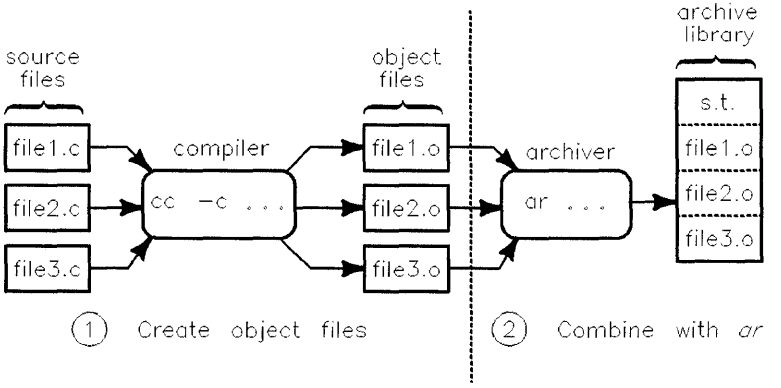


Figure 3-1. Creating an Archive Library

By default, `ar` creates the archive library in the current working directory. You can then link this library with your programs by specifying it on the command line, just like an object file.

What Does an Archive File Contain?

An archive library file consists of three 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 location, size, and other information for each routine or data item contained in the library
3. **object modules**, one for each object file specified on the `ar` command line

Object modules appear in the archive in the same order in which they were 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 /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`, etc. In other words, module names are the same as the names of the object files from which they were created.

Creating an Archive Library: An Example

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 in Figure 3-2 through Figure 3-4:

3

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

Figure 3-2. length.c—Routine to Convert Length Units

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

Figure 3-3. volume.c—Routine to Convert Volume Units

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

Figure 3-4. mass.c—Routine to Convert Mass Units

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               All the .o modules are included; it worked.
volume.o
mass.o
```

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 `/lib` or `/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 5.

Replacing, Adding and Deleting Object Modules

Occasionally you may want to replace an object module in a library, add an object module to a library, or to 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                            volume.o is gone.
mass.o
```

Summary of ar Keys

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. Table 3-1 defines some useful *keys* and their modifiers.

Table 3-1. Useful ar Keys

Key	Description
t	Display a **t**able of contents for the *archive*.
v	Display **v**erbose output.
d	**D**elete the *modules* from the *archive*.
r	**R**eplace 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*.
u	Used with the **r**, this modifier tells **ar** to replace only those *modules* with creation dates later than those in the *archive*.
x	E**x**tracts 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.
f	Truncate **f**ile names to 14 characters before comparing with file names in the archive, which are already truncated to 14 characters. Useful with long file names.

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-r--r--   265/    20     230 Feb  2 17:19 1990 length.o
rw-r--r--   265/    20     228 Feb  2 16:25 1990 mass.o
rw-r--r--   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
```

Where to Put Archive Libraries

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:

3

- in the `/lib` or `/usr/lib` directory
- in the `/usr/local/lib` or `/usr/contrib/lib` directory

Using /lib or /usr/lib

Since the linker, by default, searches `/lib` and `/usr/lib` for libraries, you might want to put the libraries here. Placing a library here eliminates your having to type the entire library path name each time you compile or link. The drawbacks of putting the libraries in these directories are:

- It typically takes super-user (system administrator) privileges to write the files into these directories.
- HP-UX system libraries reside here, so you should take care not to overwrite them.

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

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

The `/usr/local/lib` library typically contains libraries created locally—by programmers on the system; `/usr/contrib/lib` contains libraries supplied with HP-UX but not supported by Hewlett-Packard. 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.

4

Creating Shared Libraries

As discussed in Chapter 2, HP-UX provides many useful libraries of routines you can call from your programs. You can also create your *own* libraries. There are two kinds of libraries you can create—archive and shared. This chapter describes how to create *shared* libraries with the `ld` command. Specifically, it discusses:

- creating position-independent code (PIC)
- creating a shared library with `ld`
- updating a shared library
- version control
- where to put shared libraries
- linking with other libraries to create library dependencies (Series 700/800 only)
- improving performance of shared libraries

For details on creating *archive* libraries, see Chapter 3. For details on linking shared libraries with programs, and for a summary of the tradeoffs between using shared versus archive libraries, see Chapter 5.

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).

4 The `+z` (or `+Z`) option forces the compiler to generate PIC object files.

Example

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 combine (link) it with other PIC object files to create a shared library, as discussed in "Creating the Library with ld" later in this chapter.

+z versus +Z

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. You would discover this condition when creating a shared library with the `ld` command. In such cases, `ld` displays an error message, telling you to recompile the library with `+Z`.

Compiler Support for +z and +Z

The `+z` and `+Z` options work only on these compilers:

Series 300/400 computers	C and FORTRAN
Series 700/800 computers	C, FORTRAN, and Pascal

Shared Libraries with Debuggers, Profilers, and Static Analysis

As of the HP-UX 9.0 release, debugging of shared libraries is supported. For details on how debug shared libraries, refer to *HP-UX Symbolic Debugger User's Guide.*

Profiling (with `prof` and `gprof`) and static analysis are not allowed on shared libraries. If you need to profile a library, use the archive version.

Creating the 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/write 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
```

Once the executable is created, the library should not be moved because the absolute path name of the library is stored in the executable. For details, see "Shared Library Location" later in this chapter.

For details on linking shared libraries with your programs, see Chapter 5.

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 re-link 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 fixed 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.

Version Control

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.

4 **Version numbers** allow a shared library to have multiple versions of an object module.

The Version Number Compiler Directive

A version number can be assigned to any module in a shared library. It 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: `#pragma HP_SHLIB_VERSION "`*date*`"` (the quotes are optional)

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 (`90` for 1990) or a full year specification (`1990`). 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`.

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. Figure 4-1 shows the new `length.c` file.


```
#pragma HP_SHLIB_VERSION "11/92" /* date is November 1992 */
/*
 * 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, float cmstr)     /* convert in to cm */
{
           :        /* build "cmstr" */
        return(in * 2.54);
}
           :        /* other length conversion routines */
```

Figure 4-1. length.c—Length-Conversion Routines; New Version

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 volume.o mass.o length.o   Relink the library.
```

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 5.

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 04/92, `file2.o` on 05/92, and `file3.o` on 07/92, it would be best to give all the modules the same version date, say 07/92.

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 is was completed: 04/92 for `file1.o`, 05/92 for `file2.o`, and 07/92 for `file3.o`. You then build the final library on 07/92 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 05/92, 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!

Shared Library Location

You can place shared libraries in the same locations as archive libraries (see "Where to Put Archive Libraries" in Chapter 3). Again, this is typically `/usr/local/lib` and `/usr/contrib/lib` for application libraries, and `/lib` and `/usr/lib` 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.

As of 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 via 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 details on the use of these options, refer to the section "Linking a Program with Shared Libraries" in Chapter 5.

Shared Library Dependencies (Series 700/800 Only)

On Series 700/800 systems, 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 **supporting libraries**. When you load such a library, all its supporting libraries are loaded too. For example, suppose you create a library named `libdep.sl` using the command:

4

```
$ 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 supporting 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 supporting libraries loaded?
- Where are all the supporting 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 supporting libraries are loaded. The following algorithm is used:

```
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.
```

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 supporting 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` → `libB`

`libD` → `libQ`, `libB`

`libP` → `libA`, `libD`, `libQ`

Shown below are the steps that would be 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 marked, skip `B`, traverse `Q`
8. `Q` is marked, skip `Q`
9. **load** `D`
10. mark `A`, **load** `A`
11. **load** `P`

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`

4

The same behavior occurs for libraries that are explicitly loaded with `shl_load`, but without the `BIND_FIRST` modifier (see Chapter 8 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*>

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.

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 `-h` option. 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 and Exporting Symbols (-h and +e)" in Chapter 5).

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 "Linker Option Files (-c file)" in Chapter 5.)

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(s)
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.

Setting Shared Library Permissions to Non-Writable (Series 700/800 Only)

On Series 700/800 systems, 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 (Series 700/800 Only)

Normally, the Series 700/800 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 instead of the data space. This results in one copy of the constant data being shared among all processes that use the library.

4

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`".

5

Linking and Running Programs

This chapter describes how to use the linker, `ld`, to create executable programs. It describes the use of many powerful linker options that change characteristics of the executable. This chapter also describes what the operating system does when you run a program. Specifically, this chapter describes how to

- specify link libraries
- choose an archive or shared library
- link with shared libraries
- hide and export symbols in a shared library or program
- specify multiple linker options in files
- migrate to shared libraries from archive libraries
- generate shared executables
- generate demand-loaded executables
- strip symbol table information from executables
- change a program's attributes with `chatr`
- dynamically link and load object modules

This chapter does not cover detailed reference information on the linker. For such information, refer to *ld*(1); in the *HP-UX Reference*.

Linker Overview

The HP-UX linker, `ld`, produces a single executable file from one or more input object files. 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`.

Compiler-Linker Interaction

As described in Chapter 2, 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 on a Series 700 workstation produced the output below:

5

```
$ cc -Aa -v main.c func.c -lm
cc: CCOPTS is not set.
main.c:
/lib/cpp.ansi main.c /tmp/ctmAAAa10102 -D__hp9000s700 \
  -D__hp9000s800 -D__hppa -D__hpux -D__unix -D_PA_RISC1_1 \
  -A -I /usr/include
cc: Entering Preprocessor.
/lib/ccom /tmp/ctmAAAa10102 main.o -O0 -v -Aa
func.c:
/lib/cpp.ansi func.c /tmp/ctmAAAa10102 -D__hp9000s700 \
  -D__hp9000s800 -D__hppa -D__hpux -D__unix -D_PA_RISC1_1 \
  -A -I /usr/include
cc: Entering Preprocessor.
/lib/ccom /tmp/ctmAAAa10102 func.o -O0 -v -Aa
cc: LPATH is /lib/pa1.1:/usr/lib/pa1.1:/lib:/usr/lib
/bin/ld /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 linker, `/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.

The crt0.o Startup File

Notice in the previous example that the first object file on the linker command line is `/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. It also calls the main program: it calls the routine `_start` in `libc`, which in turn calls the main program as a function. On Series 300/400 computers, it calls the main program directly, without calling `_start`.

If the `-p` profiling option is specified on the 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`. For details on startup files, see *crt0*(3).

On Series 300/400 FORTRAN, the startup file is `frt0.o` instead of `crt0.o`. Also, the profiling startup files are `mfrt0.o` (if compiled with `-p`) and `gfrt0.o` (if compiled with `-G`).

Entry Point

The entry point is the location at which execution begins in the `a.out` file. It is defined in `crt0.o`. On Series 300/400 computers, the entry point is defined by the symbol `_start` in `crt0.o`. On Series 700/800 computers, the entry point is defined by the symbol `$START$` in `crt0.o`.

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 segments reside in the file? For details on the format of this file, see *a.out*(4).

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
```

Renaming the a.out File

5 To override the default name of `a.out`, use the `-o` option. For example, the following `ld` command creates an executable named `sum` from the object files `/lib/crt0.o` and `sum.o`:

```
$ ld -o sum /lib/crt0.o sum.o -lc -lm
```

Specifying Linker Options with the LDOPTS Environment Variable

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 syntax.
$ export LDOPTS

$ setenv LDOPTS "-v -t"             C shell syntax.
```

Thereafter, the following commands would be equivalent

```
$ ld /lib/crt0.o -u main prog.o -l -c
$ ld -v -t /lib/crt0.o -u main prog.o -l -c
```

Specifying Libraries (-l)

The `-l` option tells `ld` the libraries to search in to find global definitions. Its usage is:

`-l`*x*

where *x* is a character string denoting the library in which `ld` should search for global definitions. Only the part of the library name following `lib` needs to be specified with the `-l` option. For example, to specify `libc`, use `-lc`; to specify `libm`, use `-lm`.

By default, `ld` searches for the specified libraries in `/lib` and `/usr/lib`, in that order. The default order can be changed with the `LPATH` environment variable or the `-L` option, described below.

Note

On Series 700 computers, you can use the `-L` option to direct the linker to search for the faster PA1.1 libraries as follows:

```
-L/lib/pa1.1 -L/usr/lib/pa1.1
```

For details, see "Selecting Faster Libraries (Series 700/800 Only)" in Chapter 2.)

You can also use the `LPATH` environment variable (described later in this section) to do the same thing. In fact, this is how the C and FORTRAN compilers cause the linker to search the appropriate libraries for a particular architecture.

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.

Overriding the Default Linker Search Path (LPATH)

The `LPATH` environment variable allows you to specify which directories `ld` should search. If `LPATH` is *not* set, `ld` searches the default directories `/lib` and `/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=/lib:/usr/lib:/usr/local/lib   Korn and Bourne shell syntax.
$ export LPATH

$ setenv LPATH /lib:/usr/lib:/usr/local/lib   C shell syntax.
```

Augmenting the Default Linker Search Path (-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 /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 /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`.

Choosing Archive or Shared Libraries (-a and -l:)

If both an archive and shared version of a particular library reside in the same directory, `ld` links against the shared version. For example, `libc.a` and `libc.sl` both reside in `/lib`; so by default, `ld` uses `libc.sl`. 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.

There are two ways to select archive or shared libraries—with the `-a` option and the `-l:` option.

Using the -a Option

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

```
        ⎧ archive        ⎫
        ⎪ shared         ⎪
-a      ⎨ 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`	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.
`-a archive_shared`	(Series 700/800 only.) 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`	(Series 700/800 only.) This is the same as `-a default`.

For example, to link with the shared `libcurses` but the archive `libm` and `libc`, use this sequence of `-l` and `-a` options:

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

The following sequence of `-l` and `-a` options causes the linker to use the archive version of `libcurses` and the shared versions of `libm` and `libc`:

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

Using the -l: Option (Series 700/800 Only)

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 against. For instance, `-l:libm.a` causes the linker to link against archive library `/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 toggle the state of the `-a` option.

For instance, suppose you use the `LDOPTS` environment variable (see "Specifying Linker Options with the LDOPTS Environment Variable") 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`. If a particular library is available only as shared or archive, you can use `-l:` to ensure that the linker will always link against this library, regardless of the setting of the `-a` option in the `LDOPTS` variable.

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

```
$ ld /lib/crt0.o -u main prog.o -L/usr/mylibs \
    -l:libfoo.a -lc -lm
```

5

Linking a Program with Shared Libraries

When linking with shared libraries, there are some special considerations that don't apply to archive libraries. This section discusses these.

Exporting Symbols from the Main Program (-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 in Chapter 8). 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 the section "Hiding and Exporting Symbols (-h and +e)" later in this chapter.

Library Location and the Dynamic Loader (dld.sl)

An incomplete executable contains a list of absolute path names of the shared libraries searched at link time. When a program begins execution, it attaches these shared libraries. This activity is actually performed by the **dynamic loader**, which is activated by the startup code in `crt0.o`.

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, suppose that during development, a program is linked with the shared library `liblocal.sl` in your current working directory (say, `/users/hyperturbo`):

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

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.

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

Moving Libraries after Linking

As of the HP-UX 9.0 release, 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. There are two ways to specify this information:

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

5

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, on Series 700/800 computers, it can be enabled for such libraries with the `-l` option to the `chatr` command (see "Changing a Program's Attributes with chatr").

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.

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 /lib/pa1.1 directory; but through a combination of SHLIB_PATH settings and +b options, the dynamic loader ends up loading versions found in /lib instead. If this happens, make sure that SHLIB_PATH and +b are set in such a way as to avoid such conflicts.

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 /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 /lib/crt0.o +b : -L/app/lib prog.o -lfoo -lm -lc
```

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.

Mixing +b and +s

5 If a program is linked with both +b and +s, the dynamic loader builds a path list according to the order in which the options were specified. For example, if +b is specified before +s, the dynamic loader will use the path list specified by +b and append the path list specified by SHLIB_PATH.

Note No special provisions related to security issues are taken for programs that use *setuid*(2) or *setgid*(2) to change process permissions. The builder of such programs must ensure that users cannot substitute their own library on a search path and gain undesirable privileges. Since dynamic library searching is *not* the default behavior, this is not considered a security hole in the program development environment; rather, it is the responsibility of the program builder.

The Path List and the shl_load Routine

If a library is loaded with shl_load (see Chapter 8), the dynamic loader searches the path list only if the DYNAMIC_PATH flag is specified in the shl_load call and the program has also been linked with either +b or +s.

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

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.

Forcing Immediate Binding (-B immediate)

You might also 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.

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 /lib/crt0.o prog.o -lc -lm
```

Nonfatal Shared Library Binding (-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 /lib/crt0.o prog.o -B nonfatal -B immediate -lm -lc
```

5 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. If a program attempts to call or access a nonexistent symbol, it is a fatal error.

Restricted Shared Library Binding (-B restricted) (Series 700/800 Only)

The Series 700/800 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.

Hiding and Exporting Symbols (-h and +e)

The -h and +e options allow you to hide and export symbols. **Hiding** a symbol makes the symbol a local definition, accessible only from the object module or library in which it is defined. **Exporting** a symbol makes the symbol a global definition, which can be accessed by any other object modules or libraries. The syntax of the -h and +e options is:

-h *symbol*
+e *symbol*

The -h option hides *symbol*; any other global symbols remain exported unless hidden with -h. 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. For example, suppose you want to build a shared library from an object file that contains the following symbol definitions (displayed by the nm command):

5

```
$ 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.
```

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 later under "Linker Option Files (-c file)". 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.

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.

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.

5

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 /lib/crt0.o prog.o -lm -lc
```

With libraries that are explicitly loaded via `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 the Main Program (-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 /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.

Linker Option Files (-c file)

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
  :
```

5

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
```

Migrating 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.

Library Path Names

As discussed previously in "Library Location and the Dynamic Loader (dld.sl)", `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.

As of the HP-UX 9.0 release, you can circumvent potential problems arising from having the library at a different location at run time (see the earlier section "Library Location and the Dynamic Loader (dld.sl)").

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 Series 300/400 linker treats shared libraries in other respects as if they were archive libraries, in so far as this affects resolution rules.
- The Series 700/800 linker treats shared libraries more like object files.

As a consequence of the second rule on Series 700/800 computers, 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* on Series 700/800 computers. 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.

5

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 (e.g., 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.

On Series 300/400 computers, do not place assembly language labels inside data structures because the assemblers assume that the inner labels delimit new data structures; thus, the linker is free to split the data structure up in memory and to move the pieces around. The Series 300/400 assembler provides the `internal` pseudo-op to keep such internal labels from breaking up data structures.

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.

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.

Text and Data Segment Restrictions (Series 300/400 Only)

When creating a shared library or executable program, the Series 300/400 linker assumes that anything defined in the data segment is, in fact, data, and anything defined in the text segment is, in fact, executable machine code. If a program calls a procedure defined in a shared library data segment or accesses data in a text segment, it will probably dump `core`.

Startup Code (crt0.o)

To support shared libraries, `/lib/crt0.o` (and `/lib/frt0.o` on Series 300 FORTRAN) was modified at the HP-UX 8.0 release. Applications that use a startup routine other than `crt0.o` should be linked with the `-a archive` option, as they will not work with shared libraries.

5

A related issue on Series 300/400 computers is the default program entry point. In previous HP-UX releases, the default entry point was text location zero, which normally corresponds to the symbol `_start` in `crt0.o`. To support shared libraries, the location of the default entry point was moved; it immediately follows a set of tables at the beginning of the text segment. The symbol `_start` still identifies the entry point, and the `a_entry` field of the `a.out` file header still gives the correct address, but existing code that relies on an entry point of text location zero is likely to fail.

Note also that on Series 300/400 computers, address 0 is still guaranteed to contain the value 0, so NULL pointer dereferencing still returns 0 with shared libraries. However, it is non-portable and risky programming practice to depend on this.

Version Control

The shared library version control scheme presented in Chapter 4 is robust. If all the rules are followed correctly, there is little chance of a change made to a library affecting existing executables adversely. Here are some guidelines to keep in mind when making changes to a library:

- When creating the first version of a shared library, 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.
- 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 value or size represents an incompatible change.
- Any function that is changed to take advantage of an incompatible change in another module should be considered incompatible.
- 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.

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 `chrooted` environment, you must have a copy of the dynamic loader, `dld.sl`, and all required shared libraries in the correct locations.

Debugger Limitations

As of the HP-UX 9.0 release, shared libraries can be debugged just like archive libraries with few exceptions. For details on debugging shared libraries, refer to *HP-UX Symbolic Debugger User's Guide.*

Profiling Limitations

Profiling (with the `prof` and `gprof` commands and the `monitor` library function) is only possible on a contiguous chunk of 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, re-link the application with the archive version of the library, using the `-a archive` option.

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 "Magic Numbers").
- Determine where to begin execution of the program—that is, the entry point—usually in `crt0.o`.
- If the program was linked with 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.

For details on `exec`, see the *exec*(2) page in the *HP-UX Reference*.

Magic Numbers

Recorded with each executable program is a **magic number** 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` (Series 300/400/700 only.) 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`. Table 5-1 shows which linker option to use to specifically set the magic number.

Table 5-1. 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"). 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 *How HP-UX Works: Concepts for the System Administrator*.

Shareable Executables vs Shared Libraries

Shared executables—that is, executables whose magic number is `SHARE_MAGIC` or `DEMAND_MAGIC`—*are distinct from* shared libraries. With shared executables, sharing occurs at the level of the `a.out` file's text (code) only. With shared libraries, sharing occurs for any shared libraries the `a.out` file attaches. The following examples should help clarify this difference.

Figure 5-1 shows virtual memory usage for three processes whose `a.out` file is a shared executable. Virtual memory usage is decreased because each process shares the text segment.

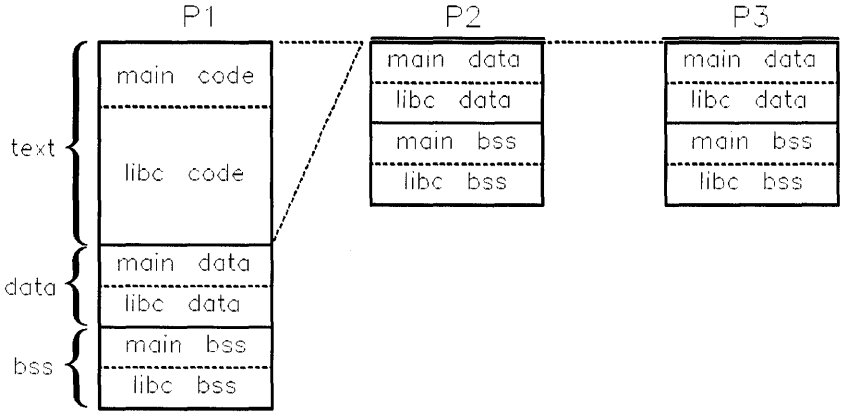


Figure 5-1. Archive Libraries with One Shared Executable

Now consider virtual memory usage if *different* `a.out` files are run with archive libraries only. Figure 5-2 shows virtual memory usage when two copies of `vi` and `ls` run simultaneously.

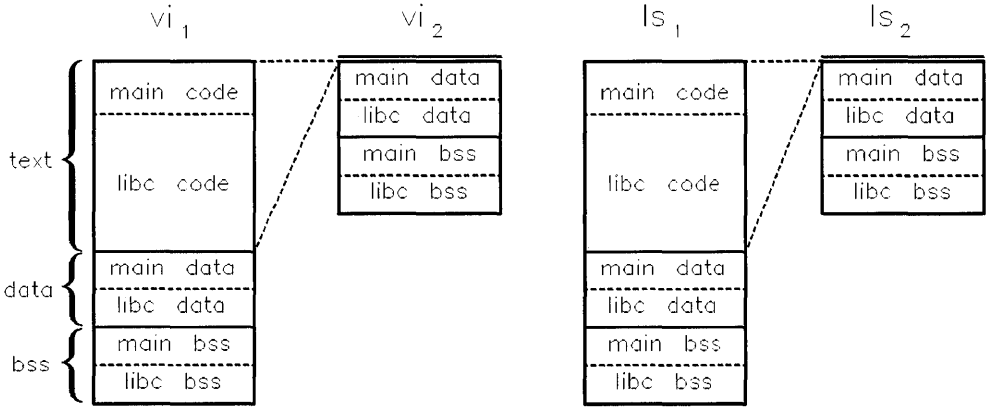


Figure 5-2. Archive Libraries with Two Shared Executables

Notice that although considerable sharing is attained, still *more* sharing could be attained if `libc` could by shared by all processes. And that is exactly what shared libraries do. Figure 5-3 shows virtual memory usage when `vi` and `ls` are linked with the shared `libc`. Imagine the further memory savings that result as more executables are linked with shared libraries.

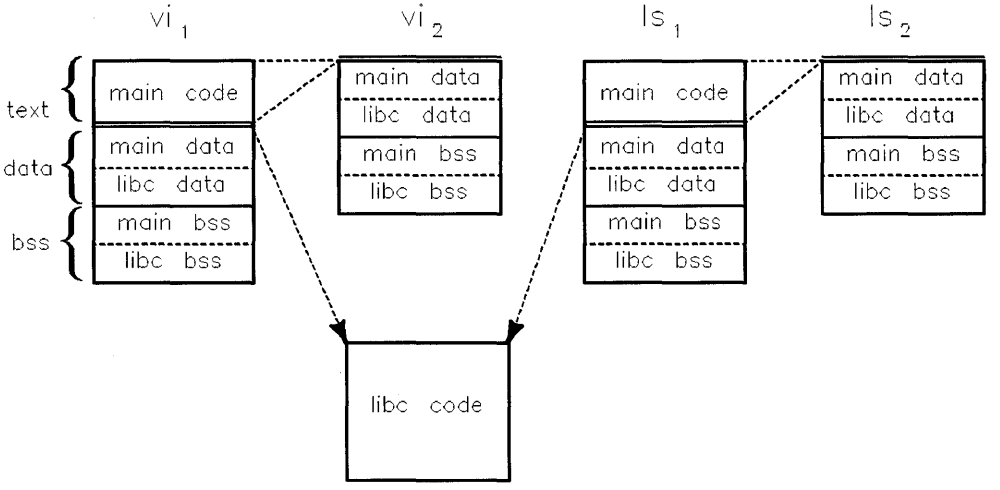


Figure 5-3. Shared Libraries with Shared Executables

Changing a Program's Attributes with chatr

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. Table 5-2 summarizes the options you can use to change various attributes.

Table 5-2. Changing Executable Attributes with chatr

To do this ...	Use option ...
Set the file's magic number to `SHARE_MAGIC`.	`-n`
Set the file's magic number to `DEMAND_MAGIC`.	`-q`
Series 700/800-Only Options	
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`[1]
Disable run-time use of the path list specified with the `+b` option at link time.	`+b disable`
Enable the use of the `SHLIB_PATH` environment variable to perform run-time path list lookup of shared libraries.	`+s enable`[1]
Disable the use of the `SHLIB_PATH` environment variable to perform run-time path list lookup of shared libraries.	`+s disable`
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*
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*

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

Stripping Symbol Table Information from the Output File

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 (such as `xdb`). You can get the same results by running the `strip` command on an executable (see *strip*(1)).

`-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 on Series 700/800 computers. Note, also, that these options can also be used when generating shared libraries without affecting shareability.

Dynamic Linking (-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 8). Nevertheless, some users will want to use this dynamic linking, which has a long history of use prior to shared libraries.

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:

5

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 (zero out) 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. On Series 700/800 computers, 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. Required only on Series 700/800; this is the default behavior on Series 300/400.
`-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.

On Series 700/800 computers, 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).

5

On Series 300/400 computers, there is only one header structure at the start of the file: `struct exec` (defined in `<a.out.h>`). The useful members of this structure are:

`.a_text`	Size of the text segment.
`.a_data`	Size of the data segment.
`.a_bss`	Size of the bss segment.
`.a_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.

On Series 700/800, 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.

On Series 300/400, the file location of the text segment is defined by the `TEXT_OFFSET(`*filhdr*`)` macro, defined in `<a.out.h>`. When passed the name of

the `exec` structure, `TEXT_OFFSET` returns the offset of the text segment in the file. Since the text and data segments are contiguous, you can determine the total size by adding the `.a_text` and `.a_data` members. This one contiguous block should be read into the address determined in Step 2.

Step 6: Clear (zero out) the bss segment.

On both architectures, 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.

On Series 700/800, 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.

On Series 300/400, the end of the data segment can be determined by adding the starting address (obtained in Step 2) to the `.a_text` and `.a_data` members of the `exec` structure. The size of the bss is defined by the `.a_bss` 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.

On Series 700/800 computers, an assembly language routine named `flush_cache` is used (see "The flush_cache Function" at the end of this chapter). You must assemble this routine separately (with the `as` command) and link it with the main program.

On Series 300/400 computers, use the *cachectl*(3C) function to do this.

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. On Series 700/800, the entry point's address is stored in the `.exec_entry` member of the `som_exec_auxhdr` structure. On Series 300/400, it is stored in the `.a_entry` member.

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 these concepts, the rest of 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`.

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`.

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

Before seeing the program's source code, it may help to see how the program and the various object files were built. Figure 5-4 shows the Makefile used to generate the various files.

```
CFLAGS = -Aa -D_POSIX_SOURCE
dynprog:        dynprog.o dynload.o
# S300/400 compile line:
#       cc -o dynprog dynprog.o dynload.o -Wl,-a,archive
# S700/800 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

# Must create flush_cache.s on S700/800:
flush_cache.o:
        as flush_cache.s
```

Figure 5-4. Makefile Used to Create Dynamic Link Files

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`	*Series 700/800 Only.* 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 flush_cache.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
```

Here are some things to note about the Makefile:

- 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`).
- Because Series 700/800 computers must link with a special assembly language routine (`flush_cache`), the compile lines for Series 300/400 and Series 700/800 systems are different. In Figure 5-4, the lines for the Series 300/400 compile line are commented out. If you were to run this on a Series 300/400 system, you would have to remove the comments and comment out the lines for Series 700/800.

(For details on using **make**, refer to *make*(1) and Chapter 13.)

Source for dynprog

Figure 5-5 shows the C source for the **dynprog** program. Notice how C preprocessor **#ifdef** directives are used to conditionally compile code for Series 300/400 or Series 700/800. In particular, symbol names on Series 300/400 begin with an underscore, but don't on Series 700/800.

```
#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";          /* name of this executable     */
const char * obj_files = "file1.o file2.o"; /* name of .o files to combine */
const char * dest_file = "dynobj.o";         /* name of .o file to load     */
#ifdef __hp9000s800                          /* next, define entry pt name  */
const char * entry_pt  = "foo";              /* no _ prefix on s700/800     */
#endif
#ifdef __hp9000s300
const char * entry_pt  = "_foo";             /* S300/400 requires _ prefix  */
#endif

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 addresses   */
/*
   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):
 */
#ifdef __hp9000s800
  nl[0].n_name = "bar";          /* S700/800 does not require _ prefix   */
  nl[1].n_name = "counter";
#endif
#ifdef __hp9000s300
  nl[0].n_name = "_bar";         /* S300/400 requires _ prefix           */
  nl[1].n_name = "_counter";
#endif
  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);
}
```

Figure 5-5. dynprog.c—Example Dynamic Link and Load Program

file1.o and file2.o

Figure 5-6 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`.

```
/*****************************************************************************
 * 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 through 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;
```

Figure 5-6. Source for file1.c and file2.c

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`. Figure 5-7 shows the `#include` directives that must appear at the start of this file. Notice that Series 300/400 and Series 700/800 systems use different header file definitions (as described in Step 4 at the start of this section).

5

```
#include <stdio.h>
#include <stdlib.h>
#include <nlist.h>
#ifdef __hp9000s800
#  include <filehdr.h>          /* S700/800 uses different header file  */
#  include <aouthdr.h>          /* definitions than the S300/400        */
#  define PAGE_SIZE 4096        /* S700/800 memory page size            */
#endif
#ifdef __hp9000s300
#  include <a.out.h>            /* S300/400 header definitions          */
#  include <sys/cache.h>        /* S300/400 needs this for cachectl(3C) */
#endif
```

Figure 5-7. Include Directives for dynload.c

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(s) 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 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. (On Series 700/800 systems, the address of the space is adjusted to begin on a memory page boundary—that is, a 4096-byte boundary.)

Figure 5-8 shows the source for this 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             */

#ifdef __hp9000s800
  struct som_exec_auxhdr file_hdr;      /* file header for S700 */
  unsigned int tdb_size; /* size of text, data, and bss combined */
#endif
#ifdef __hp9000s300
  struct exec file_hdr;  /* file header for S300                  */
#endif
/* ------------------------------------------------------------------
 * STEP 1:  Pre-link the destination module so we can get its size:
 *          (The -R option need not be specified at this time.)
 */
  sprintf(cmd_buf, "/bin/ld -a archive -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 file header for dest_file; add them together to determine size:
 */
  if ((destfp = fopen(dest_file, "r")) == NULL) {
    fprintf(stderr, "error opening %s to get bss size\n", dest_file);
    exit(1);
  }
```

```
#ifdef __hp9000s800
  /*
   * On S700/800, must seek past after SOM "header" to get to the
   * desired "som_exec_auxhdr":
   */
  if (fseek(destfp, sizeof(struct header), 0)) {
    fprintf(stderr, "error seeking to header for %s\n", dest_file);
    exit(1);
  }
#endif
  if (fread(&file_hdr, sizeof(file_hdr), 1, destfp) <= 0) {
    fprintf(stderr, "error reading header from %s\n", dest_file);
    exit(1);
  }
#ifdef __hp9000s800
  space = file_hdr.exec_tsize + file_hdr.exec_dsize + file_hdr.exec_bsize
    + 2 * PAGE_SIZE;     /* allow for page-alignment of data segment */
#endif
#ifdef __hp9000s300
  space = file_hdr.a_text + file_hdr.a_data + file_hdr.a_bss;
#endif
  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);
#ifdef __hp9000s800
  /*
   * Make sure allocated area is on page-aligned address on S700/800:
   */
  if (addr % PAGE_SIZE != 0) addr += PAGE_SIZE - (addr % PAGE_SIZE);
#endif
  return((void *) addr);
}
```

Figure 5-8. C Source for alloc_load_space Function

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.

5

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. On Series 700/800, read this information into a `som_exec_auxhdr` structure, which starts immediately after a `header` structure. On Series 300/400, read this information into an `exec` structure, which begins at the start of the file.
3. Read the text and data segments into the area allocated by `alloc_load_space`. On Series 700/800, read the text and data segments separately. On Series 300/400, read them as one contiguous block.
4. Initialize (fill with zeros) the bss, which starts immediately after the data segment.
5. Flush text from the data cache before execution. On Series 700/800, use the `flush_cache` routine (see "The flush_cache Function" later in this chapter). On Series 300/400, use the *cachectl*(3C) routine.
6. Return a pointer to the entry point, specified by the `-e` option in Step 1.

```
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               */

#ifdef __hp9000s800
  struct som_exec_auxhdr file_hdr;            /* file header for S700 */
  unsigned int tdb_size;      /* size of text, data, and bss combined */
#endif
#ifdef __hp9000s300
  struct exec file_hdr;       /* file header for S300                 */
#endif
/* ------------------------------------------------------------------
 * 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 and read its 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);
  }
#ifdef __hp9000s800
  /*
   * On S700/800, get header information from "som_exec_auxhdr" struct, which
   * is after SOM header.
   */
#endif
  if (fread(&file_hdr, sizeof(file_hdr), 1, destfp) <= 0) {
    fprintf(stderr, "failed reading file header: %s\n", dest_file);
    exit(1);
  }
/* ------------------------------------------------------------------------
 * STEP 3:  Read the text and data segments into the buffer area:
 */
#ifdef __hp9000s800
  /*
   * On S700/800, read text and data separately.  First load the text:
   */
  if (fseek(destfp, file_hdr.exec_tfile, 0)) {
    fprintf(stderr, "error seeking start of text in %s\n", dest_file);
    exit(1);
  }
  if ((fread(file_hdr.exec_tmem, file_hdr.exec_tsize, 1, destfp)) <= 0) {
    fprintf(stderr, "error reading text from %s\n", dest_file);
    exit(1);
  }
  /*
   * Now load the data:
   */
  if (fseek(destfp, file_hdr.exec_dfile, 0)) {
    fprintf(stderr, "error seeking start of data in %s\n", dest_file);
    exit(1);
  }
```

```
  if ((fread(file_hdr.exec_dmem, file_hdr.exec_dsize, 1, destfp)) <= 0) {
    fprintf(stderr, "error reading data from %s\n", dest_file);
    exit(1);
  }
#endif
#ifdef __hp9000s300
  /*
   * On S300/400, load text and data as one contiguous block:
   */
  if (fseek(destfp, TEXT_OFFSET(file_hdr), 0)) {
    fprintf(stderr, "error seeking start of text/data in %s\n", dest_file);
    exit(1);
  }
  if (fread((void *)addr, file_hdr.a_text + file_hdr.a_data, 1, destfp) <= 0) {
    fprintf(stderr, "error loading %s\n", dest_file);
    exit(1);
  }
#endif
  fclose(destfp);                /* done reading from module file */
/* ---------------------------------------------------------------------
 * STEP 4:  Zero out the bss (uninitialized data segment):
 */
#ifdef __hp9000s800
  bss_start = file_hdr.exec_dmem + file_hdr.exec_dsize;
  bss_size  = file_hdr.exec_bsize;
#endif
#ifdef __hp9000s300
  bss_start = addr + file_hdr.a_text + file_hdr.a_data;
  bss_size  = file_hdr.a_bss;
#endif
  memset(bss_start, 0, bss_size);
```

```
/* ------------------------------------------------------------------------
 * STEP 5:  Flush the text from the data cache before execution:
 */
#ifdef __hp9000s800
  /*
   * The flush_cache routine on S700/800 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 = (file_hdr.exec_dmem - file_hdr.exec_tmem) +
    file_hdr.exec_dsize + file_hdr.exec_bsize;
  flush_cache(addr, tdb_size);
#endif
#ifdef __hp9000s300
  /*
   * On S300/400, call the system library routine cachectl(3C):
   */
  cachectl(CC_FLUSH, 0, 0);
#endif
/* ------------------------------------------------------------------------
 * STEP 6:  Return a pointer to the entry point specified by -e:
 */
#ifdef __hp9000s800
  entry_pt_addr = (unsigned int) file_hdr.exec_entry;
#endif
#ifdef __hp9000s300
  entry_pt_addr = (unsigned int) file_hdr.a_entry;
#endif
  return ((void *) entry_pt_addr);
}
```

Figure 5-9. C Source for dyn_load Function

The flush_cache Function (Series 700/800 Only)

On Series 300/400 systems, the `cachectl` function (see *cachectl*(3C)) can be used to flush text from the data cache before execution. Since there is no similar routine on Series 700/800, you must create one. Figure 5-10 shows the assembly language source for such a function.

```
; flush_cache.s
;
; Routine to flush and synchronize data and instruction caches
; for dynamic loading
;
; Copyright Hewlett-Packard Co. 1985,1991
;
; 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
```

Figure 5-10. Assembly Language Source for flush_cache Function

6

Profile-Based Optimization and Data Access Optimization

This chapter describes two kinds of optimizations that are performed with the linker:

- data-access optimizations
- profile-based optimization (PBO)

Note These optimizations are available only on Series 700/800 computers.

Optimizing Access to Data (Series 700/800 Only)

On Series 700/800, the linker supports the `-O` option, which optimizes references to data. (For readers familiar with Series 700/800 assembly code, this optimization involves removing unnecessary `ADDIL` instructions from the object code.) For example, the following `ld` command results in a smaller, faster executable:

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

Invoking -O from the Compile Line

The compilers automatically call the linker with the `-O` option *if* compiler optimization level 3 is selected. For example, the following `cc` command invokes full compiler optimization as well as linker optimization:

```
$ cc -o prog +O3 prog.c
```

`+O3` *invokes* `-O` *for* `ld`

If invoked with `+O3`, the compilers generate object code in such a way that the linker can better optimize the code. Thus, the linker does a better job of optimizing code that was compiled with `+O3`.

6

Incompatibilities with other Options

The `-O` option is incompatible with these linker options:

`-b` The `-O` option has no effect on position-independent code, so `-O` is not useful when building shared libraries with `ld -b`.

`-A` Dynamic linking is incompatible with optimization.

`-r` Relocatable linking is incompatible with optimization.

`-D` Setting the offset of the data space is incompatible.

Also, `-O` is incompatible with symbolic debugging (as are any other compiler optimizations).

The linker issues a warning when such conflicts occur. If you require any of these features, do not use the `-O` option.

Profile-Based Optimization (Series 700/800 Only)

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. (A basic block is 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.)

This functionality was first available in the 8.05 release of HP-UX on Series 700 computers. At that time, it was known as **feedback-directed positioning**, since repositioning of procedures based on profile data was the only optimization performed. More powerful optimization capabilities were added at the 9.0 release, as well as support on Series 800 computers.

This section describes

- when to use PBO
- how to use PBO
- instrumenting the application
- gathering profile data for the application
- optimizing based on profile information
- management of the profile database files
- a simple example of using PBO
- restrictions and limitations of PBO
- compatibility with 8.05 PBO

Note The compiler interface to PBO is currently supported only on 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.

Not all applications will benefit from PBO. Nevertheless, two types of applications may benefit greatly from PBO:

- *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. Compilers, editors, database managers, and user interface managers are examples of such applications.

Of course, the best way to determine whether PBO will improve an application's performance is to try it.

6

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.

Instrumenting (+I/-I)

Prior to the 9.0 release, instrumentation was performed solely by the linker. As of the 9.0 release, the compiler and linker work together to instrument the code. Although you can still use just the linker to perform PBO, the best optimizations result if you use the compiler as well; this section focuses on this approach.

To instrument an application (with C and FORTRAN), compile the source with the `+I` 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. (I-SOM is an acronym for "Intermediate code-System Object Module.")

After creating an I-SOM file for each source file, the compiler invokes the linker as follows:

1. Instead of using the startup file `/lib/crt0.o`, the compiler specifies a special startup file named `/lib/icrt0.o`.
2. The compiler passes the `-I` option to the linker, causing it to place instrumentation code in the resulting executable.
3. The compiler passes the `-Fb` option, which tells the linker which code generator to use to compile the I-SOM files.

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 default optimizations (`-O`), and to see verbose output (`-v`):

```
$ cc -v -o sample.inst +I -O sample.c
/lib/cpp sample.c /tmp/ctm123
/lib/ccom /tmp/ctm123 sample.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o sample.inst sample.o -lc \
     -Fb /usr/lib/uccom
```

Pay particular attention to the linker command line (`/bin/ld ...`). Notice that the application is linked with `/lib/icrt0.o`, the `-I` option is given, and the code generator `/usr/lib/uccom` is specified.

The Startup File icrt0.o

Prior to the 9.0 release, the `icrt0.o` startup file did not exist. Instead, you linked your application with the file `/lib/measure.o`, which redefined the system `exit` function (see *exit*(2)) to write out profile data. A different approach is used at the 9.0 release.

At the 9.0 release, the `icrt0.o` startup file contains a function that writes out profile data; `/lib/measure.o` is no longer needed. The `icrt0.o` startup file uses the `atexit` system call to register this function to be called when the application exits.

Note `atexit` allows a fixed number of functions to be registered from a user application; therefore, applications linked with `-I` will have one less `atexit` call available. 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:

6

```
$ cc -c foo.c
$ cc -v -o foobar -O +I bar.c foo.o
/lib/cpp bar.c /tmp/ctm456
/lib/ccom /tmp/ctm456 bar.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o foobar bar.o foo.o \
     -Fb /usr/lib/uccom
```

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
/lib/cpp reg.c /tmp/ctm432
/lib/ccom /tmp/ctm432 reg.o -O2 -I
$ cc -v -o foobar -O +I bar.c foo.o
/lib/cpp bar.c /tmp/ctm456
/lib/ccom /tmp/ctm456 bar.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o foobar bar.o foo.o \
     -Fb /usr/lib/uccom
```

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
/lib/cpp bar.c /tmp/ctm352
/lib/ccom /tmp/ctm352 bar.o -O2 -I
/lib/cpp foo.c /tmp/ctm456
/lib/ccom /tmp/ctm456 foo.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o foobar bar.o foo.o \
     -Fb /usr/lib/uccom
```

Specifying a Code Generator to the Linker (-Fb)

As discussed in "Looking "inside" a Compiler" in Chapter 2, a compiler driver invokes several phases. On Series 700/800, 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 for C is `/usr/lib/uccom`, while the code generator for FORTRAN is `/usr/lib/uf77pass1`. To see how `-Fb` is used, refer to the previous examples.

Note Since the code generation phase is delayed until link time with PBO, linking can take much longer than usual when using PBO. And 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.

Choosing Input Data

For best results from PBO, use representative input data when running an instrumented program. Input data that tests infrequent corner cases or error conditions usually is not as good to use when profiling a program. In other words, run the instrumented program with input data that closely resembles the way the program is used in the user's environment. This results in the optimizer focusing its efforts in 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

6

When an instrumented program terminates with the *exit*(2) system call, special code in the `icrt0.o` startup 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.

Note

The `flow.data` file will not be updated if either of the following occurs:

- The process does not terminate.
- The process terminates without a call to *exit*(2).

There are many things that can cause a process to terminate without calling `exit`. For instance, the process aborts due to an unexpected signal, or the program calls *exec*(2) to replace itself with another program.

If `flow.data` does not exist, the program creates it; if `flow.data` exists, the program updates the profile data. (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 later in "Specifying a Different flow.data File with FLOW"DATA".)

As an example, suppose you have an instrumented program named `prog.inst`, and two representative input files named `inp1` and `inp2`. Then the following lines would create a `flow.data` file:

`$ prog.inst < inp1`	*Redirect input from* `inp1`.
`$ ls flow.data`	*Was* `flow.data` *created?*
`flow.data`	*Yes.*
`$ prog.inst < inp2`	*Now* `flow.data` *is updated to include profile data from the second input set.*

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.

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 re-linked since it last created profile data. In this case, the following error message will be issued:

program: `Cannot update counters. Program 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.

If there does not appear to be any activity in the `flow.data` file, the process attempting to obtain the lock gives up after a short period of time. This may happen when a program that holds the lock terminates abnormally. 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 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 via 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, `+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 the `-Fb` option to instruct the linker which code generator to use to 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
/bin/ld /lib/crt0.o -P -u main -o prog prog.o -lc \
     -Fb /usr/lib/uccom
```

Notice how the program is now linked with `/lib/crt0.o` instead of `/lib/icrt0.o` since 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 behavior of using a `flow.data` file 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
/bin/ld /lib/crt0.o -P +df prog.prf -o prog prog.o -lc \
     -Fb /usr/lib/uccom
```

Note that the +df option overrides the effects of the FLOW_DATA environment variable (see "Specifying a Different flow.data File with FLOW"DATA").

Specifying a Different flow.data File 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 /users/darraj/projectX/prog.data instead of flow.data, set FLOW_DATA:

```
$ FLOW_DATA=/users/darraj/projectX/prog.data
$ export FLOW_DATA
```

Bourne and Korn shell

```
$ setenv FLOW_DATA /users/darraj/projectX/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
/bin/ld /lib/crt0.o -P -u main -o foobar foo.o bar.o -lc \
     -Fb /usr/lib/uccom
```

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 < inp1                Profile it, storing the data under the
$ prog.inst < inp2                name prog.inst.
$ cc +P -o prog.opt prog.o        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 the database file 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.o
```

Like the `+df` option, the `+pgm` option is passed directly to the linker.

Selecting an Optimization Level with PBO

6

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 when you instrumented the application.

PBO has the greatest impact when it is combined with level 2 or greater optimizations (`-O`/`+O2` or `+O3`). For instance, this compile command combines level 2 optimization with PBO:

```
$ cc -v -O +I -o prog prog.c
/lib/cpp prog.c /tmp/ctm123
/lib/ccom /tmp/ctm123 prog.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o prog prog.o -lc \
     -Fb /usr/lib/uccom
```

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
/bin/ld /lib/crt0.o -P -u main -o prog prog.o -lc \
     -Fb /usr/lib/uccom
```

A Simple Example

Suppose a user wants 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.

First, the user compiles the application for instrumentation and level 2 optimization:

```
$ cc -v -o sample.inst +I -O sample.c
/lib/cpp sample.c /tmp/ctm123
/lib/ccom /tmp/ctm123 sample.o -O2 -I
/bin/ld /lib/icrt0.o -I -u main -o sample.inst sample.o -lc \
     -Fb /usr/lib/uccom
```


At this point, the user has an instrumented program called **sample.inst**. The user has two representative input files to use for profiling, **input.file1** and **input.file2**. Suppose that the user now executes 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 database. The second invocation increments the counters for **sample.inst** in the **flow.data** file. Then, the user moves the **flow.data** file to a file named **sample.data**.

To perform profile based optimizations on this application, the user needs to re-link the program as follows:

```
$ cc -v -o sample.opt +P +pgm sample.inst +df sample.data sample.o
/bin/ld /lib/crt0.o -P +pgm sample.inst +df sample.data -u main \
      -o sample.opt sample.o -lc -Fb /usr/lib/uccom
```

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`.

Restrictions and Limitations of PBO

This section describes restrictions and limitations you should be aware of when using PBO.

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.

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 instrumented I-SOM files? For example, this could occur if you change source code, re-instrument the code, but don't gather new profile data for the re-instrumented code.

In such a 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:

`profile-based optimization: control flow structure for procedure` *name* `changed`

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.

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:

`-b` When used with the `-P` option, the `-b` option builds an optimized shared library from I-SOM files that were compiled with the `+z` or `+Z` option. However, the resulting shared library will contain only object code, *not* I-SOM code. If specified with the `-I` option, the `-b` option is ignored and the linker generates a warning message.

`-r` The `-r` option works with both `-I` and `-P`. However, it produces an object file, *not* an I-SOM file.

`-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 8.05 PBO

Procedure-level repositioning using the linker option `-I` and `-P` was introduced in the Series 700 8.05 release. The 9.0 release has introduced features that are incompatible with the 8.05 release:

- The `FLOW_DATA_DIR` environment variable, which specified an alternate directory for reading and writing the `flow.data` file, has been replaced by the `FLOW_DATA` environment variable, which specifies an alternate file name for the profile data file.
- There are differences in the `flow.data` file's internal format:
  - New instrumented applications cannot update `flow.data` files that were built with the old format.
  - Similarly, old applications cannot modify profile database files created in the new format.

  The application will exit and an error message will be issued if either of the above is attempted.
- The linker *will* accept old `flow.data` files with the `-P` option. However, only procedure-level repositioning will be performed using this data; additional optimizations available from the new format will not be possible.
- At the 8.05 release, a program stored its profile data using the name it had when it was created by the linker. Even if you renamed the application, it would still store its data under the name it was given at link time. Now the program stores its data under its the basename specified at run time; the link name is no longer used.

7

Position-Independent Code

This chapter discusses

- relocatable object code
- position-independent code (PIC)
- PIC generated by compilers for Series 700/800 computers
- PIC generated by compilers for Series 300/400 computers

Throughout this chapter, examples of PIC are shown in assembly 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.

Note Before reading this chapter, you should have a good understanding of virtual memory concepts and memory management on HP-UX. These topics are covered in detail in the book *How HP-UX Works: Concepts for the System Administrator*.

What Is Relocatable Object Code?

Relocatable object code is 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.

For example, if you write a program that references the external variable `errno`, the object code created by the compiler contains only a reference to the symbol `errno`. Only when this object code is run through the linker does the reference to `errno` change (relocate) to an absolute address in virtual memory, say `0x40009000`. Similarly, for a call to a function, say `sum_n`, the relocatable object references the symbol corresponding to the start of the `sum_n` function; the linker assigns the absolute virtual address for this symbol at link time (relocation).

Therefore, all function and variable references in an `a.out` file must reside at a specific position within the process's address space at run time. That is, a process depends on all data and routines residing at a specific position at run time.

Note that relocatable object code *does not contain* **physical addresses.** Physical addresses refer to exact locations in physical memory. Relocatable 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. (Virtual memory management on HP-UX is described in detail in *HP-UX System Administrator Concepts.*)

Because relocatable object code may contain virtual addresses, the HP-UX program loader, `exec`, must always load the code 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 relocatable object code that does *not* contain absolute virtual addresses. This is crucial to being usable in a shared library.

In order for the relocatable object code in a shared library to be fully sharable, it must not depend on its position in the virtual address space of any particular process. The relocatable object code for a shared library may be attached at different points in different processes, so it must work independent of being located at any particular position in a process's virtual address space. (Thus 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. PIC is different on Series 300/400 and Series 700/800 architectures, as explained in the following sections.

Series 700/800 Position-Independent Code

To be position-independent on Series 700/800 computers, 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. 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.

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
```

The remainder of this section describes how the Series 700/800 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 Series 700/800 assembly language, refer to the *Assembly Language Reference Manual*.

7

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. `%19` 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
```

7 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.

7

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 `$$dyncall` must be used when calling a routine via a procedure label. The linker sets the second-to-last bit in the procedure label to flag this as a special PLT procedure label. The `$$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 `/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).

Series 300/400 Position-Independent Code

A shared library comprises several PIC object modules combined by the linker, `ld`. Object modules may contain unresolved references and require relocation, but relocation of text (code) is done when the object modules are combined with `ld`. The text of a shared library should contain no absolute virtual addresses requiring further relocation. To ensure that no absolute virtual addresses remain within the text segment, all subroutine calls and data references use indirect or PC-relative addressing modes.

When invoked with `+z` or `+Z`, the C and FORTRAN compilers generate such code. The remainder of this section describes how the compilers generate PIC for the following addressing situations:

- branches
- subroutine calls
- data references

You can use these guidelines to write assembly language programs that generate PIC object code. For details on Series 300/400 assembly language, refer to the book *HP-UX Assembler and Tools.*

Branches

Branches (both conditional and unconditional) generated by Series 300/400 compilers always transfer to a target within the same function as the instruction itself. For direct branches to a label, the `bra` instruction (or a conditional equivalent), which takes a displacement rather than an absolute address, is generated:

```
# PIC direct branch
bra.l   L1
```

For `switch` statements in C and computed `GOTO`s and multiple returns in FORTRAN, an index register is loaded with a displacement from a switch table based on the controlling expression. A PC-relative code sequence accesses the switch table, which resides in the text segment. A PC-relative `jmp` is then issued using the index register:

```
        # PIC switch statement:
        mov.l   switch_expression,%d0
        lea.l   (L1,%pc,%za0),%a0
        mov.l   (0,%a0,%d0.l*4),%d0
        jmp     L2(%pc,%d0.l)
L2:
        lalign  4
L1:
        long    L3-L2
        long    L4-L2
L3:
        # code for first case
L4:
        # code for second case
```

For FORTRAN assigned `GOTO` statements, PIC requires a PC-relative load of the label address.

```
        # PIC assigned goto
        lea.l   (L1,%pc,%za0),%a0
        jmp     (%a0)
```

Subroutine Calls

If the caller and subroutine are in the same module, then a `bsr` instruction, which takes a displacement rather than an absolute address, transfers control directly between the two. If the subroutine is not in the same file as the caller, the call must be resolved indirectly through a vector which can be referenced directly by the shared library text and can be initialized at run time by the dynamic loader. This vector, called the **procedure linkage table** (**PLT**), contains absolute addresses and is not sharable. It is inserted into the data segment of a shared library by the linker.

Each PLT performs an absolute branch to its target. The subroutine call code in the text performs a **bsr** to the appropriate PLT entry. The assembler emits a special RPLT relocation which is resolved by the linker to give the displacement from the point of call to the PLT entry for the symbol:

```
# PIC intra module function call
bsr.l   _foo                    # PC relative displacement
# PIC external function call
bsr.l   _foo                    # RPLT relocation
```

Data References

A **data linkage table** (**DLT**) resolves all data references. This table, like the PLT, is constructed by the linker and placed in the shared library data segment. Each entry contains the absolute address of a data item. The prologue of each PIC function loads the absolute base address of the DLT using a statically determined PC-relative displacement. Each data reference then accesses the item indirectly through a table entry. The assembler emits a special RDLT relocation record which is resolved by the linker to an offset from the base of the table:

```
      # PIC prologue
L1:
      mov.l   &DLT,%a0                  # PC relative relocation
      lea.l   L1(%pc,%a0.l),%a2
      # PIC data reference
      mov.l   _foo(%a2),%a0             # RDLT relocation
      mov.l   (%a0),%d0
```

Note that this offset is a 16-bit signed offset. If the linkage table is larger than 32K, then the more expensive

```
      mov.l   (_foo,%a2,%za0),%a0       # RDLT relocation (long)
```

must be generated. The **+Z** option produces code for the larger tables. The linker emits a warning if you try to build a shared library that requires **+Z** but the code was not compiled that way.

The fpa_loc Symbol and PIC

The symbol `fpa_loc`, used when programming the 98248 floating point accelerator in assembly language, is treated specially by the assembler. If you use the 98248 FPA, the `%a2` register should not be used as the DLT base because the FPA uses it. And the instruction normally used to load `fpa_loc` into `%a2` should remain:

```
lea fpa_loc, %a2
```

Do *not* change it to the PIC-style reference:

```
mov.l fpa_loc(DLT_base), %a2
```

8

Shared Library Management Routines

Normally, when creating an executable program, you specify on the command line any libraries the program needs. Such libraries are loaded when the program begins execution; this is known as **implicit loading**.

Occasionally it is not possible to know what libraries a program will need at run time, so you cannot specify the libraries at link time. Instead, the program must load the required libraries at run time.

For example, suppose you write a graphics program. The program must work with any graphics device (display, plotter, printer) that it might run on, including any device that might be supported in the future. One way to ensure that the program works with any supported graphics device is to create a shared library of routines for each graphics device. Then, at run time, determine which device the program is running on, load the appropriate shared library, and call routines from the library. Loading a library at run time is known as **explicit loading**.

This chapter describes how to

- write and compile programs using shared library management routines
- explicitly load a shared library
- call routines and reference data of an explicitly loaded shared library
- get information on currently loaded shared libraries
- get descriptor information for a shared library
- define or redefine a shared library symbol
- unload a shared library
- initialize a shared library

Linking with Shared Library Routines

The shared library management functions described in this chapter reside in the library `libdld.sl`. The *shl_load*(3X) page in the *HP-UX Reference* describes them in detail.

Here are the shared library routines:

`shl_load`	Explicitly loads a shared library.
`shl_findsym`	Finds the address of a global symbol in a shared library.
`shl_get`	Gets information about currently loaded libraries.
`shl_gethandle`	Gets descriptor information about a loaded shared library.
`shl_definesym`	(Series 700/800 only.) Adds a new symbol to the global shared library symbol table.
`shl_getsymbols`	(Series 700/800 only.) Returns a list of symbols in a shared library.
`shl_unload`	Unloads a shared library.

To use these functions, a program must be compiled or linked with the command-line option `-ldld`.

If a program uses explicitly loaded libraries that reference symbols defined in the program, link the program with the `-E` option. The `-E` option ensures that all global symbols needed by the library are exported from the program.

Since the compilers do not pass `-E` to the linker, you must use the `-Wl` option to pass `-E` to the linker:

```
$ cc -Aa prog.c -Wl,-E -ldld
```

8

Shared Library Header File (dl.h)

The 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 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 chapter, all examples are given in C. To learn how to call these routines from FORTRAN or Pascal, refer to the inter-language calling conventions described in the *HP-UX Portability Guide*.

Explicitly Loading a Shared Library

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: Each process that uses a shared library gets a copy of the library's data. 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 a shared library initializer has been declared for an explicitly loaded library, it will be called after the library is loaded. For details, see "Declaring an Initializer for a Shared Library" later in this chapter.

To explicitly load a shared library, use the `shl_load` routine.

shl_load Syntax

```
shl_t shl_load( const char * path,
                int flags,
                void * address )
```

path A null-terminated character string containing the path name of the shared library to load.

8

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.

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.
`BIND_NOSTART`	Causes the dynamic loader to *not* call the initializer (even if one is declared for the library). This will also inhibit a call to the initializer when the library is unloaded. See "Declaring an Initializer for a Shared Library" later in this chapter.
`BIND_RESTRICTED`	(Series 700/800 Only) Causes the search for a symbol definition to be restricted to those symbols that were visible when the library was loaded.

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. On Series 700/800 computers, this argument is currently ignored; mapping a library at a user-defined address is not currently supported.

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);
```

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. For details on the use of +b and SHLIB_PATH, see "Library Location and the Dynamic Loader (dld.sl)" in Chapter 5.

BIND_RESTRICTED Modifier (Series 700/800 Only)

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.

shl_load Return Value

If successful, shl_load returns a **shared library handle** of type shl_t. 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 or another library could not be found. On getting this return value, a program should terminate immediately, as this indicates that the program's symbol bindings are in an inconsistent state.

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*.

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 "Accessing Routines and Data in Explicitly Loaded Libraries".

shl_load Example

Figure 8-1 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 Figure 8-2.

8

```
#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;
}
```

Figure 8-1. load_lib—Function to Load a Shared Library

Accessing Routines and Data in Explicitly Loaded Libraries

To call a routine or access data in an explicitly loaded library, first get the address of the routine or data with `shl_findsym`.

shl_findsym Syntax

```
int shl_findsym( shl_t * handle,
                 const char * sym,
                 short type,
                 void * value )
```

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 later). *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 these values (defined in `<dl.h>`):

`TYPE_PROCEDURE`	Look for a function or procedure.
`TYPE_DATA`	Look for a symbol in the data segment (e.g., variables).
`TYPE_UNDEFINED`	Look for *any* symbol.

value A pointer in which **shl_findsym** stores the address of *sym*, if found.

shl_findsym 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. On getting this return value, a program should terminate immediately.

`EINVAL` The specified *handle* is invalid.

Using shl_findsym to Call a Routine

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

Using shl_findsym to Access Data

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

Figure 8-2 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 Figure 8-1) to load the library.

Remember that on Series 300/400 computers, linker symbols begin with an underscore, but Series 700/800 linker symbols do not. The **#ifdef** statements in the following program allow it to compile and run successfully on both architectures.

```
#include <stdio.h>      /* contains standard I/O defs        */
#include <stdlib.h>     /* contains getenv definition        */
#include <dl.h>         /* contains shared library type defs */
/*
 * Define symbols appropriately for the architecture:
 */
#ifdef  __hp9000s300
#define GOPEN   "_gopen"
#define GCLOSE  "_gclose"
#define MOVE2D  "_move2d"
#define DRAW2D  "_draw2d"
#define MAXX    "_maxX"
#define MAXY    "_maxY"
#endif
#ifdef  __hp9000s800
#define GOPEN   "gopen"
#define GCLOSE  "gclose"
#define MOVE2D  "move2d"
#define DRAW2D  "draw2d"
#define MAXX    "maxX"
#define MAXY    "maxY"
#endif
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            */
}
```

Figure 8-2. Load a Shared Library and Call Its Routines and Access Its Data

Shown below is the compile line for this program, along with the commands to set SHLPATH appropriately before running the program. (Of course, this example assumes you have created `libgrphdd.sl`.) Notice that `load_lib()`, defined in Figure 8-1, is compiled along with this program:

```
$ cc -Aa -o shl_findsym shl_findsym.c load_lib.c -ldld

$ SHLPATH=/lib/libgrphdd.sl
$ export SHLPATH
$ shl_findsym
```

Getting Information on Currently Loaded Libraries

To obtain information on currently loaded libraries, use the `shl_get` function.

shl_get Syntax

```
int shl_get( int index,
             struct shl_descriptor **desc )
```

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 will increment all the shared library indexes by one.
- The program unloads a shared library with `shl_unload`. Any libraries following the unloaded library will 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. On Series 300/400, the name of the main program is not known, so `shl_get` uses the filename `<a.out>` for the main program.
`initializer`	A pointer to the shared library's initializer routine (see "Declaring an Initializer for a Shared Library"). 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`.

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.

shl_get 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.

shl_get Usage

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.

shl_get Example

The function `show_loaded_libs` in Figure 8-3 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.

```
#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);
}
```

Figure 8-3. show_loaded_libs—Display Library Information

Calling this function from a C program compiled with shared `libc` and `libdld` produced the following output on a Series 700/800 computer:

```
SUMMARY of currently loaded libraries:
___library___          _tstart_    __tend__    _dstart_    __dend__
./a.out                  0x1000      0x1918  0x40000000  0x40000200
/usr/lib/libdld.sl  0x800ac800  0x800ad000  0x6df62800  0x6df63000
/lib/libc.sl        0x80003800  0x80091000  0x6df63000  0x6df85000
```

On a Series 300/400 computer, it produced this output:

```
SUMMARY of currently loaded libraries:
___library___          _tstart_     __tend__     _dstart_     __dend__
<a.out>                       0       0x1000       0x1000       0x1288
/usr/lib/libdld.sl   0x80004000   0x80005000   0x80005000   0x80006000
/lib/libc.sl         0x80006000   0x8007c000   0x8007c000   0x8009ec38
```

Getting Descriptor Information for a Shared Library

The `shl_gethandle` routine returns descriptor information about a loaded shared library.

shl_gethandle Syntax

```
int shl_gethandle( shl_t handle,
                   struct shl_descriptor **desc )
```

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.

shl_gethandle Return Value

If *handle* is not valid, the routine returns −1 and sets `errno` to `EINVAL`. Otherwise, `shl_gethandle` returns 0.

shl_gethandle Example

Figure 8-4 shows a function named `show_lib_info` that displays information about a shared library, given the library's handle.

```
#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;
}
```

Figure 8-4. show_lib_info—Display Information for a Shared Library

Defining or Redefining a Shared Library Symbol (Series 700/800 Only)

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.

shl_definesym Syntax

```
int shl_definesym( const char *sym,
                   short type,
                   long value,
                   int flags )
```

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 via *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.

shl_definesym Return Value

If successful, `shl_definesym` returns 0. Otherwise, it returns −1 and sets `errno` accordingly. See *shl_definesym*(3X) for details.

8

shl_definesym Usage

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 will always use its own definition for any data (storage) symbol, even if a more visible one is provided.
- The incomplete executable will only use a more visible code symbol if the main program itself does not provide a definition.

Retrieving Symbols Defined in a Shared Library (Series 700/800 Only)

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. Use of this routine is unnecessary for most programmers.

shl_getsymbols Syntax

```
int shl_getsymbols( shl_t handle,
                    short type,
                    int flags,
                    void * (*memfunc)(),
                    struct shl_symbol **symbols )
```

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 (e.g., variables).
`TYPE_UNDEFINED`	Retrieve *all* symbols, regardless of type.

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.

One of the following modifiers can be ORed with the `EXPORT_SYMBOLS` value:

`NO_VALUES` Do not calculate the `value` field of the `shl_symbol` structure for symbols. The `value` field will have 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 below under 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 next.

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` 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 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. On getting this value, a program should terminate immediately.

`EINVAL` The specified *handle* is invalid.

shl_getsymbols Example

Figure 8-5 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** will display 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.

```
#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;

  num_symbols = shl_getsymbols(hndl, type, flags, malloc, &symbols);
  if (num_symbols < 0) {
    printf("shl_getsymbols failed\n");
    exit(1);
  }
  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(symbols);                         /* free memory allocated by malloc */
  return num_symbols;                    /* return the number of symbols */
}
```

Figure 8-5. show_symbols—Display Shared Library Symbols

Figure 8-6 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.

```
#include <dl.h>
#include <stdio.h>
int show_syms(shl_t hndl, short type, int flags); /* prototype for show_syms */
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);
    }
}
```

Figure 8-6. show_all—Use show_symbols to Show All Symbols

The program in Figure 8-6 was compiled with the command:

```
$ cc -Aa -o show_all show_all.c show_symbols.c -ldld
```

Figure 8-7 shows partial output produced by running this command on a Series 700 system.

```
[show_all]
  Imports:
    _start                          PROCEDURE
    malloc                          PROCEDURE
    free                            PROCEDURE
    exit                            PROCEDURE
    printf                          PROCEDURE
    shl_get                         PROCEDURE
    shl_getsymbols                  PROCEDURE
      TOTAL SYMBOLS: 7
  Exports:
    errno                           STORAGE    0x4000122C
    _SYSTEM_ID                      DATA       0x40001008
    __dld_loc                       STORAGE    0x40001228
    _end                            DATA       0x40001230
    main                            PROCEDURE  0x6DF86362
      TOTAL SYMBOLS: 5
[/usr/lib/libdld.sl]
  Imports:
    errno                           STORAGE
    __dld_loc                       DATA
      TOTAL SYMBOLS: 2
```

Figure 8-7. Output of show_all Program

Unloading a Shared Library

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 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 a shared library initializer has been declared for a shared library, it will be called when the shared library is unloaded. For details, see "Declaring an Initializer for a Shared Library" later in this chapter.

shl_unload Syntax

```
int shl_unload(shl_t handle)
```

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`.

shl_unload 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.

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.

Declaring an Initializer for a Shared Library

A shared library can have an initialization routine—known as an **initializer**—that is called when the shared library is loaded or unloaded. Typically, an initializer is used to initialize a shared library's data when the library is loaded. The initializer is called for libraries that are loaded implicitly (at program startup) or explicitly (via `shl_load`).

When calling initializers for *implicitly* loaded libraries, the dynamic loader waits until all libraries have been loaded before calling the initializers. On Series 700/800, 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. On Series 300/400, it calls the initializers in the same order in which the libraries are searched for symbols. On both architectures, all initializers are called before the main program begins execution.

When calling the initializer for an explicitly loaded library, the dynamic loader waits until any dependency libraries (available on Series 700/800 only) are loaded before calling the initializers. As with implicitly loaded libraries, initializers are called in depth-first order on Series 700/800, and in the library search order on Series 300/400.

Note that initializers can be disabled for explicitly loaded libraries via the `BIND_NOSTART` flag to `shl_load`; see "Explicitly Loading a Shared Library".

Declaring the Initializer

To declare the name of the initializer, use the `+I` linker option when creating the shared library. In addition, the shared library must reference the initializer.

The +I Linker Option

The syntax of the `+I` option is:

`+I` *initializer*

initializer is the initializer's name. (On Series 300/400 systems, be sure to prefix the name with an underscore.) For example, to create a shared library named `libfoo.sl` that uses an initializer named `init_foo`, use this linker command line on Series 700/800:

```
$ ld -b -o libfoo.sl libfoo.o +I init_foo
```

On Series 300/400, use this command line:

```
$ ld -b -o libfoo.sl libfoo.o +I _init_foo
```

Referencing the Initializer from the Shared Library

Note that it is not sufficient to use `+I` to declare the initializer; the library must also contain a reference to the initializer. The actual definition of the initializer can appear in the shared library or in the main program.

For instance, suppose `init_foo` is defined in `libfoo.sl` in the preceding example. To ensure that `init_foo` is registered as the initializer, you could include the following line in the library's source:

```
void (*init_foo_ptr)() = init_foo;
```

If, on the other hand, `init_foo` is defined outside the library (say, in the main program), you would need to declare `init_foo` as an external symbol:

```
extern void init_foo();
void (*init_foo_ptr)() = init_foo;
```

The Default Initializer (Series 300/400 Only)

On Series 300/400 systems, if a routine named `_INITIALIZER` is referenced in the shared library, it is assumed to be the initializer for the shared library, and no `+I` option is required to declare it. However, for compatibility reasons, it is probably best to always use `+I __INITIALIZER` to declare such initializers.

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.

Example: An Initializer for Each Library

One way to use initializers is to define a unique initializer for each library. For instance, Figure 8-8 shows the source code for a library named `libfoo.sl` that contains an initializer named `init_foo`.

Note The examples shown in this section are all for Series 700/800 computers. The examples will also work on Series 300/400 if you prefix all symbol names with an underscore.

```
#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");
}
void (*init_ptr)() = init_foo;     /* must reference 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);
}
```

8

Figure 8-8. C Source for libfoo.sl

Note that the reference "`void (*init_ptr) = init_foo;`" ensures that `init_foo` is registered as the initializer. Here are the commands used to create `libfoo.sl` on a Series 700/800 system:

```
$ 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. Figure 8-9 shows an example program that loads and unloads `libfoo.sl`, and Figure 8-10 shows the output of running this program.

```
#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 libunits.sl and find the required symbols:
   */
  if ((hndl_foo = shl_load("libfoo.sl", BIND_IMMEDIATE, 0)) == NULL)
    perror("shl_load: error loading libunits.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 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));
  /*
   * Unload the library:
   */
  shl_unload(hndl_foo);
}
```

Figure 8-9. C Source for testlib

```
libfoo loaded
1.0in  =  2.54cm
1.0gal =  3.79l
1.0oz  = 28.35g
libfoo unloaded
```

Figure 8-10. Output of testlib

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). Figure 8-11 shows the source for such an `_INITIALIZER` function.

```
#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)                    /* find the lib's load function */
    shl_findsym(&hand, "load", TYPE_PROCEDURE, (void *) &load_unload);
  else                            /* find the lib's unload function */
    shl_findsym(&hand, "unload", TYPE_PROCEDURE, (void *) &load_unload);

  (*load_unload)();               /* call the function */
}
```

Figure 8-11. C Source for _INITIALIZER (file init.c)

Figure 8-12 and Figure 8-13 show the source for two shared libraries that have registered _INITIALIZER.

```
#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();
void (*init_ptr)() = _INITIALIZER;   /* must reference 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);
}
```

Figure 8-12. C Source for libunits.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");
}
```

Figure 8-13. C Source for libtwo.sl

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
```

Figure 8-14 shows a program that loads these two libraries.

8

```
#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);
}
```

Figure 8-14. C Source for testlib2

Here is the compiler command used to create the executable `testlib2`:

```
$ cc -Aa -Wl,-E -o testlib2 testlib2.c init.c -dld
```

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, Figure 8-15 shows the output from running `testlib2`.

```
libunits.sl loaded
libtwo.sl loaded
1.0in  =  2.54cm
1.0gal =  3.79l
1.0oz  = 28.35g
foo called
bar called
libunits.sl unloaded
libtwo.sl unloaded
```

Figure 8-15. Output of testlib2

9

Standard Input/Output Library Routines

This chapter describes how to use standard input/output library routines—that is, routines that are designated as section "3S" in the *HP-UX Reference*. The **standard input/output library** is a collection of routines that provides efficient and portable input/output services for most C programs. The standard input/output library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change. Specifically, this chapter describes:

- an overview of standard input/output routines
- input/output to standard input (`stdin`) and standard output (`stdout`)
- input/output to strings
- input/output to ordinary files
- stream status and control routines
- inter-process communication

All the examples in this chapter are written in C. Nevertheless, these routines can be called from other languages, although this normally isn't necessary as each language has an extensive set of input/output routines. For details on calling C library routines from other languages, refer to the *HP-UX Portability Guide*.

Overview of Input/Output

To call standard input/output routines, a C program must `#include` the header file `<stdio.h>`:

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

This file contains function prototypes and type definitions required for standard input/output routines.

In HP-UX terminology, files are often referred to as **streams**. Each stream has an associated **buffer**, through which input or output data is passed. When a program writes data to a stream (for example, using the `fprintf` routine), the data is actually passed to the buffer. The data can then be **flushed** from the buffer. Flushing is usually performed automatically by standard input/output routines, but sometimes you might want greater control over flushing; this is accomplished through stream status and control routines.

Before reading or writing data to a stream, a program must **open** it. When a program begins executing, HP-UX automatically opens three streams for the program: **standard input**, **standard output**, and **standard error**. These files are referred to as `stdin`, `stdout`, and `stderr`, respectively.

Typically, `stdin` corresponds to terminal keyboard input; `stdout` corresponds to terminal screen output; and `stderr` is used for displaying error messages to the terminal screen. However, all of these can be redirected to or from other sources, as described in the *Using HP-UX with HP Vue*.

The files `stdin`, `stdout`, and `stderr` are different from ordinary files in that they store small amounts of data that exists only until it is read or written. (One exception is that characters can be "pushed back" into the input stream, described later in this chapter.) Another difference is that `stdin` is a read-only file; a program cannot write to `stdin`. Similarly, a `stdout` and `stderr` are write-only files; they cannot be read.

By default, the buffers used with `stdin` and `stdout` are `_DBUFSIZ` bytes long, where `_DBUFSIZ` is a constant, defined in `<stdio.h>` as 8192. In fact, all fully buffered files use a buffer `_DBUFSIZ` bytes in length, by default. In contrast, `stderr` is not buffered; data is transferred to `stderr` one byte at a time. Due to terminal driver characteristics, data typed at the keyboard is not sent to `stdin` until Return (or its equivalent) is pressed.

A program is not limited to using only the standard input and output streams. A program can also open ordinary text files for reading, writing, or both at the same time. Directories can also be opened, but only for reading. These features are discussed later in this chapter. The next section discusses the use of routines that work with `stdin` and `stdout`; `stderr` is described later.

Input/Output Using stdin and stdout

This section describes three pairs of input/output routines that interact with `stdin` and `stdout`. They are:

- `getchar` and `putchar` for single-character input/output
- `gets` and `puts` for string input/output
- `scanf` and `printf` for formatted input/output of all types

Single-Character Input/Output

This section describes the two basic input and output routines, `getchar` and `putchar`. `getchar` is a macro defined in `<stdio.h>` which reads one character from `stdin`. Similarly, `putchar` is also a macro defined in `<stdio.h>`. `putchar` writes one character on `stdout`.

As an example, consider the following program, which simply reads `stdin` and echoes whatever it finds to `stdout`. The program terminates when it receives an at-sign (`@`) from `stdin`.

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '@')
        putchar(c);
   putchar('\n');
}
```

Why is c declared an **int** instead of a **char**? For most applications, **char** works fine. In certain cases, however, sign extension, bit shifting, and similar operations cause strange results with **chars**. Therefore, **int** is used here, and in all following examples, to be safe.

The final **putchar** statement in the program is used to output a new-line so that your shell prompt appears at the beginning of a new line, instead of at the end of the last line of output. Type it in and give it a try! Remember that your input is not available to the program until you press RETURN.

getchar and **putchar** are most useful in **filters** which are programs that accept data and modify it in some way before passing it on. Suppose you want to write a program which puts parentheses around each vowel encountered in the input. It's easy to do with these routines:

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '\n') {
      if(vowel(c)) {
         putchar('(');
         putchar(c);
         putchar(')');

      }else
          putchar(c);
   }
vowel(c)
char c;
{
   if(c=='a' || c=='A' || c=='e' || c=='E' || c=='i' || c=='I'
   || c=='o' || c=='O' || c=='u' || c=='U')
      return(1);
   else
      return(0);
}
```

The vowel test is placed in the function **vowel**, since it tends to clutter up the main program. This program terminates when it encounters a new-line.

String Input/Output

The **gets** function reads a string from **stdin** and stores it in a character array. The string is terminated by a new-line in the input, which **gets** replaces with a NULL character in the array. Its companion function, **puts**, copies a string from a character array to **stdout**. The string is terminated by a NULL character in the array, which **puts** replaces with a new-line in the output.

The simple "echo" program from the last section can be rewritten using **gets** and **puts**:

```
#include <stdio.h>
main()
{
   char line[80], *gets();

   while((gets(line)) != NULL)
      puts(line);
}
```

This program, as written, runs forever. To terminate it, press BREAK (or its equivalent). Later, when string comparison and string length routines are introduced, an intelligent termination condition can be written for this program.

Formatted Input/Output with scanf

scanf is the formatted-input library routine. Its syntax is:

scanf(*format*, [*item* [,*item*] ...]);

where *format* is a character pointer to a character string (or the character string itself enclosed in double quotes), and *item* is the address of a variable.

format specifies the format of incoming data to be read from **stdin**, and what types of data are found there. **format** is composed of two elements: conversion specifications and literal characters.

Conversion Specifications

A conversion specification is a character sequence which tells `scanf` how to interpret the data received at that point in the input. For example, if a conversion specification says "treat the next piece of data as a decimal integer", then that data is interpreted and stored as a decimal integer.

In the format, a conversion specification is introduced by a percent sign (%), optionally followed by an asterisk (*) (called the **assignment suppression character**), optionally followed by an integer value (called the **field width**). The conversion specification is terminated by a character specifying the type of data to expect. These terminating characters are called **conversion characters.**

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many, the excess items are simply ignored. The one exception occurs when the assignment suppression character (*) is used. If an asterisk occurs immediately after the percent sign (before the field width, if any), then the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list. This is useful for skipping over unwanted data in the input.

Conversion Characters

There are eight conversion characters available. Three of them are used to format integer data, three are used to format character data, and two are used for floating-point data.

The integer conversion characters are:

`d` A decimal integer is expected.
`o` An octal integer is expected.
`x` A hexadecimal integer is expected.

The character conversion characters are:

`c` A single character is expected.
`s` A character string is expected.
`[` A character string is expected.

The floating-point conversion characters are:

`e, f` A floating-point number is expected.

Integer Conversion Characters

The `d`, `o`, and `x` conversion characters read characters from `stdin` until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

For `d`, an inappropriate character is any character *except* +, −, and 0 through 9. For `o`, an inappropriate character is any character *except* +, −, and 0 through 7. For `x`, an inappropriate character is any character *except* +, −, 0 through 9, and the characters a − f and A through F. Note that negative octal and hexadecimal values are stored in their 2's complement form with sign extension. Thus, they may look unfamiliar if you print them out later (using `printf` − see below).

These integer conversion characters can be capitalized or preceded by a lower-case L (`l`) to indicate that a `long int` should be expected rather than an `int`. They can also be preceded by `h` to indicate a `short int`. The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The `c` conversion character reads the next character from `stdin`, no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a field width is specified, then the number of characters indicated by the field width are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the `c` conversion character are *not* automatically terminated with a NULL character in the array. Since all C library routines which utilize strings assume the existence of a NULL terminator, be sure you add the NULL character yourself. Otherwise, library routines are not able to tell where the string ends, and you'll get puzzling results.

The `s` conversion character reads a character string from `stdin` which is delimited by one or more space characters (blanks, tabs, or new-lines). If no field width is given, the input string consists of all characters from the first non-space character up to (but not including) the first space character. Any initial space characters are skipped over. If a field width is given, then characters are read, beginning with the first non-space character, up to the first space character, or until the number of characters specified by the field width is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

An important point to remember about the `s` conversion character is that it *cannot* be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line consisting of 10 spaces followed by "Hello, there!" (the vertical bar shows the beginning of the line but is not included in the text string):

```
|          Hello, there!
```

If you use `%c`, you get a space character. However, if you use `1s`, you get "H" (the first non-space character in the input).

The `[` conversion character also reads a character string from `stdin`. However, this character should be used when a string is *not* to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a

circumflex (^), then characters are read from **stdin** until a character is read which matches one of the characters between the brackets. If the first character is *not* a circumflex, then characters are read from **stdin** until a character *not* occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating NULL character which is added automatically.

The three string conversion characters provide you with a complete set of string-reading capabilities. The **c** conversion character can be used to read *any* single character, or to read a character string *when the exact number of characters in the string is known beforehand.* The **s** conversion character enables you to read any character string *which is delimited by space characters, and is of unknown length.* Finally, the **[** conversion character enables you to read character strings *that are delimited by characters other than space characters, and which are of unknown length.*

Floating-Point Conversion Characters

The **e** and **f** conversion characters read characters from **stdin** until an inappropriate character is encountered, or until the number of characters specified by the field width, if given, is exhausted (whichever comes first).

Both **e** and **f** expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an **E** or **e** followed by an optionally signed integer. Thus, an inappropriate character is any character *except* +, −, ., 0 through 9, E, or e.

These floating-point conversion characters can be capitalized, or preceded by a lower-case L (**l**), to indicate that a **double** value is expected rather than a **float**. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format which are *not* part of a conversion specification are **literal characters**. A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs ("%%") to get a literal percent sign.

Examples. Suppose that you have to read the following line of data:

```
NAME: Joe Kool; AGE: 27; PROF: Elec Engr; SAL: 39550
```

To get the vital data, you must read two strings (containing spaces), and two integers. You also have data that should be ignored, such as the semicolons and the identifying strings ("NAME:"). How do you go about reading this?

First, note that the identifying strings are always delimited by space characters. This suggests use of the `s` conversion character to read them. Second, you can never know the exact sizes of the NAME and PROF fields, but note that they are both terminated by a semicolon. Thus, you can use `[` to read them. Finally, the `d` conversion character can be used to read both integers.

The following code fragment successfully reads this data:

```
char name[40], prof[40];
int age, salary;
    :
scanf("%*s%*[ ]%[^;]%*c%*s%d%*c%*s%*[ ]%[^;]%*c%*s%d",\
      name,&age,prof,&salary);
```

For easier understanding, break the format into pieces:

`%*s` Reads the string "NAME:". Since an asterisk is given, the string is simply read and discarded.

`%*[ ]` Gets rid of all blanks occurring between "NAME:" and the employee's name. Note that this gets rid of one or more blanks, giving the format some flexibility.

`%[^;]` Reads all characters from the current character up to a semicolon, and assigns the characters to the array *name*.

`%*c` Gets rid of the semicolon left over after reading the name.

`%*s` Reads the next identifying string, "AGE:", and discards it.

`%d` Reads the integer age given, and assigns it to `age`. The semicolon after the age terminates %d, because that character is not appropriate for an integer value. Note that the address of `age` is given in the item list (`&age`) instead of the variable name itself. If this is not done, a memory fault occurs at run time.

`%*c` Gets rid of the semicolon following the age.

`%*s`	Reads the next identifying string, "PROF:", and discards it.
`%*[ ]`	Removes all blanks between "PROF:" and the next string.
`%[^;]`	Reads all characters up to the next semicolon, and assigns them to the character array *prof*.
`%*c`	Gets rid of the semicolon following the profession string.
`%*s`	Reads the final identifying string, "SAL:", and discards it.
`%d`	Reads the final integer and assigns it to the integer variable *salary*. Again, note that the address of *salary* is given, not the variable name itself.

Although somewhat confusing to read, this format is quite flexible, since it allows for multiple spaces between items and varying identifying strings (that is, "PROFESSION:" could be specified instead of "PROF:"). The following `scanf` call reads the same data, but is much less flexible:

```
scanf("NAME: %[^;]; AGE:%d; PROF: %[^;]; SAL: %d",\
      name,&age,prof,&salary);
```

Here, literal characters are used to exactly match the characters in the input line. This works fine if you can be sure that the data always appears in this form. If one typing variation is made, however, such as typing "SALARY:" instead of "SAL:", the `scanf` fails.

`scanf` waits for more data as long as there are unsatisfied conversion specifications in the format. Thus, a `scanf` call like

```
scanf("%f%f%f", &float1, &float2, &float3);
```

where *float1*, *float2*, and *float3* are all variables of type `float`, allows you to enter data in several ways. For example,

```
14.77 29.8 13.0
```

is read correctly by `scanf`, as is

```
14.77 RETURN
29.8 RETURN
13.0 RETURN
```

Using decimal points in floating-point data *is recommended* whenever floating-point variables are being read. However, `scanf` converts integer data

to floating-point if the conversion specification so demands. Thus, "13.0" in the previous example could have been entered as "13" with no side effects.

As a final example, suppose the following code fragment is used to read the input string "abcdef137 d14.77ghijklmnop":

```
char arr1[10], arr2[10], arr3[10], arr4[10];
float float1;
scanf("%4c%[^3]%6c%f%[ghijkl]",arr1,arr2,arr3,&float1,arr4);
```

To determine what values are stored in the variables listed, break up the format into separate conversion specifications, and see what data is demanded by each (as done before):

`%4c`	Reads four characters and assigns them to `arr1`. Thus, the string "abcd" is assigned to `arr1`. Note that an extra character, NULL, is appended to the end of the string.
`%[^3]`	Reads all characters from the current character up to the character "3". This assigns "ef1", along with an added NULL character, to the array `arr2`.
`%6c`	Reads the next six characters and stores them in the array `arr3`. Thus, "37 d14" is assigned to `arr3`, terminated by a NULL character.
`%f`	Reads a floating-point value which, due to the lack of a field width, is terminated by the first "inappropriate" character. Thus, the value ".77" is assigned to `float1`.
`%[ghijkl]`	Reads all characters up to the first character not occurring between the brackets. This stores the string "ghijkl", along with an appended NULL character, in the array `arr4`.

Note that there are some characters left in `stdin` that were not read. What happens to these characters? Any characters left unread in the input stream *remain there*. This can cause unexpected errors. Suppose that, later in the above program fragment, you want to read a string from `stdin` using `%s`. No matter what string you type in as input, it won't be read because the `%s` conversion specification is satisfied by reading "mnop"—the characters left over from the previous read operation! To solve this, always be sure you have read the entire current line of input before attempting to read the next. To fix this

in the previous `scanf` example, just add a `%*s` conversion specification at the end of the format. This reads and discards the left-over characters.

Formatted Output with printf

For output, `printf` is the companion routine to `scanf`. It enables you to output data in formatted form. Its syntax is the same as `scanf`:

`printf(`*format*, [*item* [, *item*] ...]);

format is a pointer to a character string (or the character string itself enclosed in double quotes) which specifies the format and content of the data to be printed. Each *item* is a variable or expression specifying the data to print.

`printf`'s format is similar in many respects to that of `scanf`. It is made up of conversion specifications and literal characters. As in `scanf`, literal characters are all characters that are not part of a conversion specification. Literal characters are printed on `stdout` exactly as they appear in the format.

Literal Characters

Included in the list of literal characters are **escape sequences**, which are sequences beginning with a backslash (\) which stand for other characters. The following list shows the escape sequences defined for `printf` (and `scanf`, though less frequently used):

`\b`	Backspace.
`\n`	Newline (carriage-return/line-feed sequence); output begins at the beginning of a new line.
`\r`	Carriage-return without a line-feed; output begins at the beginning of the current line (data already printed on that line is over-printed).
`\t`	Tab.
`\\`	Literal backslash.
`\`*nnn*	The character represented by the octal number *nnn* in the ASCII character set. *nnn* must begin with a zero. For example, `\007` is an ASCII BELL character, which beeps the terminal bell (if the bell function exists on the terminal).

Conversion Specifications

A conversion specification for `printf` is very similar to that of `scanf`, but is a bit more complicated. The correct sequence for the components of a conversion specification are

1. a percent sign (%), which signals the beginning of a conversion specification
2. zero or more *flags*, which affect the way a value is printed (see below)
3. an optional decimal digit string which specifies a minimum *field width*
4. an optional *precision* consisting of a dot (.) followed by a decimal digit string
5. an optional `l` (lowercase L) or `h`, indicating a `long` or `short` integer argument
6. a *conversion character*, which indicates the type of data to be converted and printed

---

Note To output an actual percent sign character, you must type two percent signs ("%%").

As in `scanf`, a one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *flags* are:

- Causes the data to be left-justified within its output field. Normally, the data is right-justified.

+ Causes all signed data to begin with a sign (+ or −). Normally, only negative values have signs.

blank Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the "blank" and "+" flags both appear, the "blank" flag is ignored.

Causes the data to be printed in an "alternate form". Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.

A *field width*, if specified, determines the *minimum* number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the – flag is specified) to fill the field. If the data is larger than the field width, the field width is simply expanded to accommodate the data. An insufficient field width *never* causes data to be truncated. If no field width is specified, the resulting field is made just large enough to hold the data.

The *precision* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

A field width or precision can be replaced by an asterisk (*). If so, the next item in the item list is fetched, and its value is used as the field width or precision. The item fetched must be an integer.

Conversion Characters

Conversion character specifies the type of data to expect in the item list, and causes the data to be formatted and printed appropriately. Integer conversion characters include:

`d` An integer *item* is converted to signed decimal. The precision, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the precision, the value is expanded with leading zeros. The default precision is one (1). A null string results if a zero value is printed with a zero precision. The **#** flag has no effect.

`u` An integer *item* is converted to unsigned decimal. The effects of the precision and the **#** flag are the same as for `d`.

`o` An integer *item* is converted to unsigned octal. The **#** flag, if specified, causes the precision to be expanded and the octal value is printed with a leading zero (a C convention). The precision behaves the same as in `d` above, except that printing a zero value with a zero precision results in only the leading zero being printed if the **#** flag is specified.

`x` An integer *item* is converted to hexadecimal. The letters `abcdef` are used in printing hexadecimal values. The **#** flag, if specified, causes the precision to be expanded, and the hexadecimal value is printed with a leading "0x" (a C convention). The precision behaves as in `d` above,

except that printing a zero value with a zero precision results in only the leading "0x" being printed if the **#** flag is specified.

`X` Same as `x` above, except that the letters `ABCDEF` are used to print the hexadecimal value, and the **#** flag causes the value to be printed with a leading `0X`.

The character conversion characters are as follows:

`c` The character specified by the `char` *item* is printed. The precision is meaningless, and the **#** flag has no effect.

`s` The string pointed to by the character pointer *item* is printed. If a precision is specified, characters from the string are printed until the number of characters indicated by the precision has been reached, or until a NULL character is encountered, whichever comes first. If the precision is omitted, all characters up to the first NULL character are printed. The **#** flag has no effect.

The floating-point conversion characters are:

`f` The `float` or `double` *item* is converted to decimal notation in **style f**—that is, in the form:

$$[-]ddd.ddd$$

where the number of digits after the decimal point is equal to the precision. If no precision is specified, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. If the **#** flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e` The `float` or `double` *item* is converted to scientific notation in **style e**; that is, in the form:

$$[-]d.ddd\ \mathtt{e}\pm ddd$$

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the precision. If no precision is given, six (6) digits are printed after the decimal point. If the precision is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the **#** flag is specified, the result

always contains a decimal point, even if no digits follow the decimal point.

`E` Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (**style E**).

`g` The `float` or `double` *item* is converted to either *style f* or *style e*, depending on the size of the exponent. If the exponent resulting from the conversion is less than −4 or greater than the precision, *style e* is used. Otherwise, *style f* is used. The precision specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the **#** flag is specified, the result always has a decimal a point, even if no digits follow the decimal point, and trailing zeros are *not* removed.

`G` Same as the `g` conversion above, except that *style E* is used instead of *style e*.

The *items* in the item list can be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are *not* required in the item list (contrast this with `scanf`'s item list). If the `s` conversion is used, a pointer to a character string must be specified.

Examples

Here are some examples of `printf` conversion specifications and a brief description of what they do:

`%d` Output a signed decimal integer. The field width is just large enough to hold the value.

`%-*d` Output a signed decimal integer. The left-justify flag (−) and the blank flag are specified. The asterisk causes a field width value to be extracted from the item list. Thus, the item specifying the desired field width must occur before the item containing the value to be converted by the `d` conversion character.

`%+7.2f` Output a floating-point value. The `+` flag causes the value to have an initial sign (+ or −). The value is right-justified in a 7-column field, and has exactly two digits after the decimal point. This conversion specification is ideal for a debit/credit column on a

finance worksheet. (If the + sign is not necessary, use the blank flag instead.)

Consider the following program, which reads a number from **stdin**, and prints that number, followed by its square and its cube:

```
#include <stdio.h>
main()
{
   double x;

   printf("Enter your number: ");
   scanf("%F", &x);
   printf("Your number is %g\n", x);
   printf("Its square is %g\nIts cube is %g\n", x*x, x*x*x);
}
```

The **g** conversion character is used so that the decision about whether or not to use an exponent is automated. Note that the item list contains expressions to calculate **x** squared and **x** cubed. Also note that the address of the variable is required in order to read a value for it, but printing requires the variable name itself.

The following program accepts a decimal integer, then prints the number, its square, and its cube in decimal, octal, and hexadecimal:

```
#include <stdio.h>
main()
{
  long n, n2, n3;

/* get value */

  printf("Enter your number: ");
  scanf("%D", &n);

/* print headings */

  printf("\n\n              Decimal     Octal     Hexadecimal\n");

 /* do the computation */

  n2 = n * n;
  n3 = n * n * n;
  printf("n itself:    %7ld   %9lo     %6lx\n", n, n, n);
  printf("n squared:   %7ld   %9lo     %6lx\n", n2, n2, n2);
  printf("n cubed:     %7ld   %9lo     %6lx\n", n3, n3, n3);
}
```

Strings are especially easy to manipulate using `printf`. The following simple program illustrates this:

```
#include <stdio.h>
main()
{
   char first[15], last[25];

   printf("Enter your first and last names: ");
   scanf("%s%s", first, last);
   printf("\nWell, hello %s, it's good to meet you!\n", first);
   printf("%s, huh?  Are you any relation to that famous\n", last);
   printf("computer programmer, Mortimer Zigfelder %s?\n", last);
   printf("No, sorry, that was my mistake.  I was thinking of\n");
   printf("O'%s, not %s.\n", last, last);
 }
```

This program shows how easily strings can be inserted in text. Try variations of your own.

Input/Output from/to Strings

Two library routines, `sscanf` and `sprintf`, enable you to read data from a string, and write data into a string. These routines behave identically to `scanf` and `printf`, respectively, except that `sscanf` reads data from a character string instead of from `stdin`, and `sprintf` writes data into a string instead of on `stdout`.

Reading Data from a String

`sscanf` enables you to read data directly from a string. The syntax for an `sscanf` call is

`sscanf(`*string*`,` *format*`,` [*item* [`,` *item*] ...]`);`

where *string* is the name of a character array containing the data to be read, and *format* and *item* are familiar terms from the previous section. Thus, the

only difference between **sscanf** and **scanf**, is **sscanf**'s *string* parameter from which data is scanned.

The following program simply reads a string of your choosing from **stdin**, stores it in the character array **string**, and prints out the first word of that string:

```
#include <stdio.h>
main()
{
   char string[80], word[25], *gets();

/* get the string */

   printf("Enter your string: ");
   gets(string);

/* get the first word */

   sscanf(string, "%s", word);
   printf("The first word is %s.\n", word);
}
```

However, **sscanf** is rarely used in this way. **sscanf** is; more often used as a means of converting ASCII characters into other forms, such as integer or floating-point values. For example, the following program uses **sscanf** to implement a five-function calculator:

```
#include <stdio.h>
main()
{
   char line[80], *gets(), op[4];
   long n1, n2;
   double arg1, arg2;

   printf("\n> ");     /* print prompt (>) and get input */
   gets(line);
      while(line[0] != 'q') {
         sscanf(line, "%*s%s", op);
         if(op[0] == '+') {
            sscanf(line, "%F%*s%F", &arg1, &arg2);
            printf("Answer:  %g\n\n", arg1+arg2);
      } else if(op[0] == '-') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1-arg2);
      } else if(op[0] == '*') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1*arg2);
      } else if(op[0] == '/') {
         sscanf(line, "%F%*s%F", &arg1, &arg2);
         printf("Answer:  %g\n\n", arg1/arg2);
      } else if(op[0] == '%') {
         sscanf(line, "%D%*s%D", &n1, &n2);
         while(n1 >= n2)
            n1 -= n2;
         printf("Answer:  %ld\n\n", n1);
      } else
         printf("Can't recognize operator: %s\n\n", op);
      printf("> ");
      gets(line);
   }
}
```

The calculator program accepts input lines having the form

value + *value*	*addition*
value - *value*	*subtraction*
value * *value*	*multiplication*
value / *value*	*division*
value % *value*	*remainder*

where *value* is any number, and any operator symbol shown can be used for the corresponding type of operation. All functions except remainder are handled internally in floating-point, but values for these functions can be typed with or without a decimal point. Values for the remainder function must not have a decimal point. There must be at least one space between each value and the operator.

Note the use of `sscanf` in this program. The entire input line is read using `gets`. Then, the different parts of the input line are read from `line` using `sscanf`. Notice that the input line is stored as an ASCII string in `line`, but portions of it are converted to floating-point or integer values, depending on the operator.

Examples of valid entries are

```
15.778 * 3.89
27 % 8
17 + 39.72
```
etc.

The program terminates when it reads a line beginning with `q`, such as "quit".

Two things differ between reading data from `stdin` and reading data from a string: First, remember that when you read data from `stdin`, the data no longer exists in `stdin`. This is *not* true for a string. Also, since the data is stored in a string, it is always there, even if that data has been read several times. Second, since the data read from `stdin` disappears as you read it, the next read operation from `stdin` always begins where the previous read operation terminated. This is *not* true when you read from a string using `sscanf`. Each successive read operation begins *at the beginning of the string*. Thus, if you want to read five words from a string stored in a character array, you must read them in a single `sscanf` call. If you try to read one word in five

separate **sscanf** calls, each call starts reading at the beginning of the string, and you end up reading the same word five times.

Writing Data into a String

The **sprintf** routine enables you to write data into a character string. Its syntax is:

sprintf(*string*, *format*, [*item* [, *item*] ...]);

string is the name of the character string into which the data is written. *format* and *item* are familiar terms from the previous discussion of **printf**. In fact, the only difference between **sprintf** and **printf** is that **sprintf** writes data into a character array, while **printf** writes data on **stdout**.

The following program acts as a "formatter" for personal data. Suppose that this program is used to provide a "friendly" user interface to gather personal data. The data received is then reformatted into a string which is passed along to another program, such as a data base maintainer. The string contains the data entered by the user, but in a form using strict field widths for the various pieces of data. The data base program requires these field widths in order for the data to be processed correctly, but there is no reason to burden the user with this requirement. This "formatter" program lets the user enter data in a convenient form (without the fixed field restrictions imposed by the data base).

```
#include <stdio.h>
main()
{
   char name[31], prof[31], hdate[7], curve[3], string[81];
   char *format = "%30s%2d%30s%61d%6s%2d%2s";
   int age, rank;
   long salary;

   printf("\nName (30 chars max): ");      /* start asking */
   gets(name);
   while(name[0] != ']') {
     printf("Age: ");
     scanf("%d%*c", &age);
     printf("Job title (30 chars max): ");
     gets(prof);
     printf("Salary (6 digits max, no comma): ");
     scanf("%D%*c", &salary);
     printf("Hire date (numerical MMDDYY): ");
     gets(hdate);
     printf("Percentile ranking (omit \"%%\"): ");
     scanf("%d%*c", &rank);
     printf("Pay curve: ");
     gets(curve);
                                        /* format string */
     sprintf(string,format,name,age,prof,salary,hdate,rank,curve);
     printf("\n%s\n", string);

     printf("\nName (30 chars max): "); /* start next round */
     gets(name);
   }
}
```

This program asks you questions to obtain typical company information such as name, age, job title, salary, hire date, ranking, and pay curve. This data is then packed into a 78-character string using **sprintf**. The string is printed on your screen in this program, but in an actual working environment, this string would probably be passed directly to the data base program. Note that **sprintf**'s format is specified as an explicit character pointer. When lengthy,

unchanging formats are used, this is often more convenient than typing the entire format string, especially if the item list is long.

As an exercise, consider the `scanf` calls in the previous program. Notice that a `%*c` conversion specification is included in the formats of the `scanf`s which are reading integer values (age, salary, rank). Why is this necessary? If you aren't sure, take the `%*c`s out of those formats, re-compile the program, run it, and note its behavior. (Remember that a new-line character terminates the read operation for `%d` and `%D` conversions, and leaves the new-line unread in `stdin`.)

Input/Output Using Ordinary Files

So far, you have been using library routines which can perform input/output only by using `stdin` and `stdout`. This section introduces routines that enable you to open existing ordinary files for reading, writing, or both, and to create ordinary files. Routines that enable you to perform input/output to and from ordinary files are also described.

Opening Ordinary Files

Before a file can be read from or written to, it must be *opened*. A file is opened using the `fopen` library routine. The syntax of an `fopen` call is:

`fopen(`*filename*`,` *type*`);`

where *filename* is a character pointer to a character string specifying the name of the file to be opened, and *type* is a character pointer to a one- or two-character string specifying the input/output operation for which the file is opened. The available *type*s are:

`r` Opens the file for reading at the beginning of the file. The file must already exist, or an error occurs.

`w` Opens the file for writing at the beginning of the file. If the file exists, its previous contents are destroyed. If the file does not exist, it is created.

`a` Opens the file for writing at the end of the file (appends data to the end of the file). If the file does not exist, it is created for writing.

`r+` Opens the file for both reading and writing, starting at the beginning of the file. The file must already exist, or an error occurs.

`w+` Opens the file for both reading and writing, starting at the beginning of the file. If the file already exists, its previous contents are destroyed. If the file does not exist, it is created.

`a+` Opens the file for both reading and writing, starting at the end of the file. If the file does not exist, it is created.

When a file is opened for an append operation (*type* `a` or `a+`), it is impossible to overwrite the existing file contents. `fseek` can be used to reposition the file pointer to any position in the file, but when output is written to the file, the pointer is disregarded. When the append operation (which begins at the end of the existing file) is completed, the file pointer is repositioned to the end of the appended output.

In exchange for a *filename* and a *type*, `fopen` opens a "pathway" between your program and the file. This "pathway" is called a **stream**. If you open the file for reading, then the stream provides one-way data transfer from the file to your program. If you open the file for writing, then data transfer flows from your program to the file. Finally, if the file is opened for both reading and writing, the resulting stream is bi-directional.

`fopen` also associates a buffer with the stream. This gives the stream the ability to store a small amount of data. By default, the capacity of the buffer is equal to `_DBUFSIZ` bytes, where `_DBUFSIZ` is a constant, defined in `<stdio.h>` as 8192.

The buffer size can be increased, decreased, or set to zero by using `setbuf` or `setvbuf`. If the buffer size is allowed to remain at default size, a maximum of `BUFSIZ` bytes of data can be present on the stream at any given time. If the buffer size is reduced to zero, then the stream can transfer only one byte at a time.

When opening a file for both reading and writing (`r+`), be sure to use `fflush` or `fseek` when switching from reads to writes (or vice versa) because, otherwise, buffering can become corrupted. For example, if a program has written to a file repeatedly and you want to start reading from the file at the current file pointer, use `fflush` to flush the output buffer before reading.

Since `fopen` takes care of all the intricacies of building a stream and allocating a buffer, all you need to know is how to find your end of the stream. `fopen` provides you with this information by returning to you a value called a *file pointer* (often called a *stream pointer*). A file pointer "points" to the newly-created stream, and keeps track of where the next input/output operation takes place (in the form of a byte offset relative to the beginning of the associated buffer).

Once you have a file pointer in your possession, you need never refer to the open file by its name again. A file pointer provides access to all the information needed by other standard input/output routines to read from or write to the file.

The following program fragment shows how the `fopen` routine is used:

```
#include <stdio.h>
main()
{
   FILE *fp;

   fp = fopen("/users/tom/bin/datafile", "r");
   if(fp == NULL) {
      printf("Can't open datafile.\n");
      exit(1);
   }
   :
}
```

This `fopen` call, if successful, opens */users/tom/bin/datafile* for reading. The file pointer name returned by `fopen` is stored in `fp`. Note that `fp`'s value is checked to see if it is NULL. This is because `fopen` returns a NULL pointer if the indicated file cannot be opened. It is good practice to check the value of a file pointer because this is the only error indication facility that `fopen` provides.

The previous example also introduces a new type declaration, `FILE`. The `FILE` declaration is defined in `<stdio.h>`. In the example above, it defines `fp` as a variable containing a file pointer. Note that explicit declarations of functions returning file pointers is unnecessary because `<stdio.h>` declares all such functions for you.

Before moving on, keep in mind that several things can stop you from successfully opening a file. First, HP-UX limits the number of files simultaneously open in a process (the limit for your system is specified in the *System Administrator Manual* supplied with your system). Remember that `stdin`, `stdout`, and `stderr` are automatically opened for you, so unless you close these files, the maximum you can explicitly open is three fewer than the system limit. Second, you must have permission to open the file for the particular *type* you have specified (this permission is granted or denied by the file's mode). Third, trying to open a non-existent file using type `r` or `r+` always fails. Fourth, if *filename* is specified incorrectly, is a non-existent directory name, or contains an intermediate component that is not a directory, the open fails. This is not a complete list, but gives you several common reasons why an attempt to open a file might fail.

fclose

`fclose` flushes the buffer associated with the specified stream, and, if the buffer was allocated automatically by the standard input/output system, frees the space allocated to that buffer. The stream is then closed, breaking the connection between your file pointer and the stream.

You may wonder why some example programs in this chapter open files but never explicitly close them. There are two reasons why this is permissible:

- First, all programs in this chapter that open files end with a call to `exit`. The `exit` system call automatically performs an `fclose` for every open file in that process.
- Second, when a program is compiled with `cc` (or `fc`, or `pc`), an `exit` call is automatically compiled into your code.

Keep in mind, however, that it is generally bad programming practice to rely on the system to close files. A program that explicitly opens a file should also explicitly close the file. If this is inconvenient, a program should at least include an `exit` call at each termination point in the program.

Single-Character Input/Output

Now that you know how to open files and obtain file pointers, you have a whole new set of input/output routines at your disposal, enabling you to perform all kinds of input/output operations. In fact, there are about three times as many available routines that utilize file pointers as there are routines that are limited to `stdin` and `stdout` only!

In this section, only those routines that read or write one character at a time are discussed. These routines are `getc`, `putc`, `fgetc`, and `fputc`. `getc` and `putc` are macros defined in `<stdio.h>` that respectively read and write a single character on the specified stream. Syntax is as follows:

`getc(`*stream*`);`

`putc(`*c*`,` *stream*`);`

where *stream* is a file pointer obtained from `fopen`, and *c* is a variable of type `char` (or `int`) indicating the character to write on the indicated stream.

Here is a simple version of the HP-UX `cat` command written using these routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int c;
   FILE *fp;

   if(argc != 2) {
      printf("Usage:  cat file\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if(fp == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   while((c = getc(fp)) != EOF)
      putc(c, stdout);
   putc('\n', stdout);

   exit(0);
}
```

This program accepts a single argument which is assumed to be the name of a file whose contents are to be printed on the user's terminal. The specified file is opened for reading, and the resulting file pointer `fp` is used in `getc` to read a character from the file. Each character read is written on `stdout` using `putc` (note that `stdout`, as well as `stdin` and `stderr`, are perfectly legal file pointers). The reading and writing loop is terminated when the constant `EOF` (defined in `<stdio.h>`) is returned from `getc`, indicating that the end of the `filesize` has been reached.

Note that `getc` and `putc` can be made to behave exactly like the `getchar` and `putchar` routines discussed earlier by specifying the appropriate file pointer. In other words,

```
getc(stdin);
```

is identical to

```
getchar();
```

and

```
putc(c, stdout);
```

is identical to

```
putchar(c);
```

Thus, the `putc` call in the previous program could just as easily have been:

```
putchar(c);
```

without altering the behavior of the program. However, if the destination of the data is somewhere other than the user's terminal, the flexibility of `putc` is required. Take, for example, the following program, which is a simple version of the HP-UX `cp` command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    int c;
    FILE *from, *to;

    if(argc != 3) {
        printf("Usage:  cp fromfile tofile\n");
        exit(1);
    }

    from = fopen(argv[1], "r");
    if(from == NULL) {
        printf("Can't open %s.\n", argv[1]);
        exit(1);
    }
    to = fopen(argv[2], "w");
    if(to == NULL) {
        printf("Can't create %s.\n", argv[2]);
        exit(1);
   }

   while((c = getc(from)) != EOF)
        putc(c, to);

   exit(0);
}
```

This program accepts two arguments. The first is the name of the file to be copied, and the second is the name of the file to be created. The first file is opened for reading, and the second file is created for writing. The data from the first file is then copied directly to the newly-created file.

The **fgetc** and **fputc** routines are actual functions, not macros. Their syntax and usage is identical to **getc** and **putc**. However, here are some distinctions between the macro and function versions of these routines to help you decide which to use:

- A function call takes time, since the function call still exists at run time. A macro call, however, takes no time at all, because the macro call is replaced with the actual code making up the macro during compilation, before run time. Thus, generally speaking, programs containing macros run faster than programs containing the equivalent function calls.
- A function's code is localized in one section of the program. Each function call causes a jump to that section to execute the function. A macro call, however, is replaced with its code everywhere that macro call appears. Thus, programs containing macro calls generally require more space than programs containing the equivalent function calls.
- The address of a function can be passed as an argument, but the address of a macro call cannot.

Given these guidelines, decide which routines to use based on your own constraints.

Character Push-Back

The `ungetc` routine enables you to push back a single character onto an input stream. This character is then returned by the next `getc` call (or equivalent).

`ungetc`'s syntax is as follows:

```
ungetc(c, stream);
```

where *c* is the character to be pushed back, and *stream* is the input stream where the push-back is to occur. Note that *c must* be the character that was last read from *stream*.

The following program simply reads one character from `stdin`, pushes it back onto `stdin`, re-reads the character, and checks to make sure that this character and the character originally pushed back are the same. A message is printed on `stdout` stating the outcome of the comparison.

```
#include <stdio.h>
main()
{
   int c1, c2;

   c1 = getchar();
   ungetc(c1, stdin);
   c2 = getchar();
   if(c1 == c2)
      printf("They're the same!\n");
   else
      printf("Oops!  They're different!\n");
}
```

One character of push-back is guaranteed as long as something has been read from the stream prior to the push-back attempt, and provided that the stream is buffered. More characters could possibly be pushed back, but determining exactly how many characters of push-back you can safely perform is quite possibly not worth the effort. However, for completeness, the following statement is included as a method for determining the number of characters of push-back available at any given time:

```
numpb = ftell(stream) % BUFSIZ + 1;
```

where `ftell` is a function discussed in a later section, *stream* is a file pointer, and `BUFSIZ` is a constant defined in `<stdio.h>` containing the size of the buffer in bytes. After execution, `numpb` contains the number of characters of push-back available at that time.

String Input/Output

The `fgets` and `fputs` routines enable you to read or write strings from or to specified streams. Their syntax is:

```
fgets(string, n, stream);

fputs(string, stream);
```

where *string* is a pointer to a character string, and *stream* is a file pointer to the input or output stream.

fgets reads a character string from the specified *stream* and stores it in the character array pointed to by *string*. **fgets** reads **n**−1 characters, or up to a new-line character, whichever comes first. If a new-line character is encountered, it is retained as part of the string (contrast this with **gets**, which replaces the new-line with a NULL character). **fgets** appends a NULL character to the string.

fputs writes the character string pointed to by *string* on the specified *stream*, stopping when a NULL character is encountered. **fputs** does *not* append a new-line character to the string when it is written. This is because **fputs** is intended for use with **fgets**, which incorporates a new-line character into the string if a new-line is encountered in the input.

The cp program written earlier can be re-written using **fgets** and **fputs**:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char c, line[256], *fgets();
   FILE *from, *to;

   if(argc != 3) {
      printf("Usage:  cp fromfile tofile\n");
      exit(1);
   }

   from = fopen(argv[1], "r");
   if(from == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }
   to = fopen(argv[2], "w");
   if(to == NULL) {
      printf("Can't create %s.\n", argv[2]);
      exit(1);
   }

   while(fgets(line, 256, from) != NULL)
      fputs(line, to);

   exit(0);
}
```

This program functions exactly like the previous version of cp above. Note that **fgets**'s return value is compared to NULL in the **while** loop, since **fgets** returns the NULL pointer when it reaches the end of its input.

This program can easily be converted to a simple **cat** command. It only requires four changes. Can you see what they are? First, change the **argc** comparison such that it reads

```
if(argc != 2) . . .
```

(You might also want to change the associated usage message!) Second, remove the *to* file pointer, since you don't need it anymore. Third, remove the block of code which uses **fopen** to open the new file, and assigns a value to *to*. Fourth, change the **fputs** call such that it reads:

```
fputs(line, stdout);
```

Here's the new **cat** command:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char c, line[256], *fgets();
   FILE *from;

   if(argc < 2) {
      printf("Usage:  cat file\n");
      exit(1);
   }

   from = fopen(argv[1], "r");
   if(from == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   while(fgets(line, 256, from) != NULL)
      fputs(line, stdout);

   exit(0);
}
```

Formatted Input/Output

Just as there are versions of **scanf** and **printf** which perform string input/output, so there are versions which enable input/output using files. **fscanf** enables you to read data of all types from a specified stream, and **fprintf** provides the capability of writing data on a stream. Their syntax is:

fscanf(*stream*, *format*, [*item* [, *item*] ...]);

fprintf(*stream*, *format*, [*item* [, *item*] ...]);

stream is a file pointer to an open stream. *format* and *item* should be familiar terms from previous discussions.

The following program illustrates the use of the **fscanf** and **fprintf** routines:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int count = 0;
   FILE *file;

   if(argc != 2) {
      fprintf(stderr, "Usage:  wdcnt filename\n");
      exit(1);
   }

   file = fopen(argv[1], "r");
   if(file == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   while(fscanf(file, "%*s") != EOF)
      count++;
   printf("Number of words found:  %d\n", count);
   exit(0);
}
```

This program, named **wdcnt** (for "word count"), counts the number of "words" in the file specified as its only argument. A word is defined as a string of non-space characters.

Note how **fprintf** is used in this program. You learned in a prior discussion that **stderr** is typically used to output error messages or warning statements. In this program, **fprintf** is used to direct error messages to **stderr**. You don't lose anything by doing this, since data written on **stderr** appears on your terminal by default. However, you gain some important flexibility. Now that error output is written on a different stream than normal output, the error output (or the normal output) can be *redirected* to another destination. For example, invoking the previous program as

```
wdcnt file1 &2> errmsgs
```

causes all output arising from erroneous conditions to be collected in the file **errmsgs**. For the **wdcnt** program, this is somewhat trivial, since the program terminates upon any error. However, for programs which output any number of warnings without terminating, this is a very useful capability. Not only does it keep normal, desired output from getting cluttered up with error messages, but it enables you to save output for later examination at your leisure. Thus, it is good programming practice to write error messages and warnings on **stderr**, and use **stdout** (or whatever your destination file is) to output normal data.

Binary Input/Output

The routines described in this section deal with data in its binary form; that is, the data is never converted to ASCII for user viewing. These routines are used to transfer raw data between two points, such as from a variable to a data file, or vice versa.

Note The alignment of members within structures can differ from one architecture to another. Thus, binary input/output can create data file incompatibilities between architectures. In C, you can get around such alignment problems with alignment **pragmas**. For details on the use of these pragmas, see the C language documentation for your system.

Two routines, **getw** and **putw**, are used to read or write an integer word (an **int**) to or from a stream, respectively. Their syntax is:

```
getw(stream);

putw(w, stream);
```

where *stream* is a file pointer to the input or output stream, and *w* is the integer word to be output by **putw**.

The following program "sorts" an existing data file containing raw integer data. The program divides this data file into two new data files; one containing integer data whose absolute value is less than or equal to 32767, the other containing data whose absolute value is larger than 32767.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }
   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }
   datale = fopen("dfle", "w");
   if(datale == NULL) {
      fprintf("Can't create dfle file.\n");
      exit(1);
   }
   datagt = fopen("dfgt", "w");
   if(datagt == NULL) {
      fprintf("Can't create dfgt file.\n");
      exit(1);
   }
   while((word = getw(dfile)) != EOF) {
      if(word <= 32767 && word >= -32767)
         putw(word, datale);
      else
         putw(word, datagt);
   }
   exit(0);
}
```

9

This program reads a word from the specified data file. If its absolute value is less than or equal to 32767, the word is written on a file called `dfle` in the

user's current directory. Otherwise, the word is written on a file called `dfgt` in the current directory.

Note that this program works only on machines that use four-byte integers. Also, the comparison between `word` and the constant `EOF` is faulty, since `EOF` is defined to be -1, a valid integer. The section entitled *Stream Status Inquiry Routines* describes standard input/output routines which fix this problem.

Both of these routines transfer four bytes at a time. Again, there is no ASCII conversion associated with these routines, so if you attempt to print the contents of a file containing integer data output by `putw`, you will get meaningless results. Note that it makes little sense to input binary data from `stdin`, as in

```
getw(stdin);
```

unless `stdin` is redirected from a file containing binary data. Using `getw` to read data from your keyboard is futile. If you type in a valid-looking integer, like "1728", `getw` reads the ASCII values of those characters and stores them as an integer. It is unlikely that *ever* get what you intended using such a method.

Two other routines, called `fread` and `fwrite`, provide much more flexible binary data input and output. Their syntax is:

```
fread((char *)ptr, sizeof(*ptr), nitems, stream);

fwrite((char *)ptr, sizeof(*ptr), nitems, stream);
```

where *ptr* is a pointer to the beginning of a block (array) of data. This argument is cast as a character pointer because these routines expect a pointer of this type. The second argument specifies the number of bytes per *unit* of data (four bytes per `int`, one byte per `char`, `x` bytes per `struct`, etc.). The C operator `sizeof` is usually used to obtain this value. The third argument, *nitems*, is an integer specifying the number of units of data to read or write. For example, if *ptr* points to the beginning of a structure, `sizeof(*ptr)` tells how many bytes make up that structure, and *nitems* tells how many structures to read. Actually, the second and third arguments above can be reversed in the argument list with no ill effects, because internally these routines simply multiply the two integers together to obtain the total number of bytes to read. Finally, `stream` is a file pointer to the input or output stream.

As an example, suppose you use a program to keep track of certain employee data where each employee is to be described in a single structure. Here is a simple program to do that:

```
#include <stdio.h>
struct emp {
          char       name[40]; /* name */
          char       job[40];  /* job title */
          long       salary;   /* salary */
          char       hire[6]   /* hire date */
          char       curve[2]  /* pay curve */
          int        rank;     /* percentile ranking */
}
#define EMPS 400               /* no.  of employees */
main()
{
     int items;
     struct emp staff[EMPS];
     FILE *data;

     data = fopen("/usr/lib/employees/empdata", "r");
     if(data == NULL) {
        fprintf(stderr, "Can't open employee data file.\n");
        exit(1);
     }

     items = fread((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Insufficient data found.\n");
        exit(1);
     }

     fclose(data);
     archive("/usr/lib/employees/empdata");

/* Employee information processing goes here. */

     . . .processing goes here. */
```

```
/* Processing is done.  Write out new employee records. */

     data = fopen("/usr/lib/employees/empdata", "w");
     if(data == NULL) {
        fprintf(stderr, "Can't create new employee file.\n");
        exit(1);
     }

     items = fwrite((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Write error!\n");
        exit(1);
     }

     exit(0);
}
archive(filename)
char *filename;
{
     . . .processing goes here. */
}
```

This program reads the employee information contained in the binary file */usr/lib/employees/empdata*. The data in this file consists of concatenated streams of bytes describing each employee of a certain 400-employee company. The bytes are written such that, when read correctly, the bytes correspond exactly with the `emp` structure defined in the program. The `staff` array is an array of structures containing one structure for each employee.

In the `fread` call, the `sizeof(staff[0])` expression returns the number of bytes in the *emp* structure. Since the same number of bytes are in each employee structure, any element of the `staff` array could have been specified as the `sizeof` argument; `staff[0]` is used in this example. (By counting the number of bytes in each structure member, you can get an approximation of the number of bytes returned by the `sizeof` operator: 40 + 40 + 8 + 6 + 2 + 4 = 100 bytes. This may vary due to padding performed by a programming language, or by machine architecture.) Specifying `EMPS` as the `nitems` argument tells `fread` to read 400 such structures. Thus, 100 x 400 = 40000

bytes are read, filling in the information for the members of each structure contained in the **staff** array.

The **archive** function is not shown here, but simply saves the old employee information in **empdata** in an employee information archive of some kind. After the information is archived, the **empdata** file is overwritten with the new, updated employee information.

A new routine, called **fclose**, is introduced here. **fclose** simply closes the stream associated with the file pointer specified. This is necessary in order to re-open the file for writing. Once it is open for writing, **fwrite** is used to overwrite its previous contents with the new data.

One final note about these two routines: they return the number of items of data which have been read or written. Thus, you can compare this number with whatever you specified for *nitems* to see if everything you wanted read or written actually was. This return value is used twice in the above program to flag probable read and write errors.

The **fread** and **fwrite** routines can be made to read *any* type of data. The following examples show various **fread** calls used to read different types of data:

- to read a long integer:

```
long nint;
fread((char *)&nint, sizeof(nint), 1, stream);
```

- to read an array of 100 long integers:

```
long nint[100];
fread((char *)nint, sizeof(nint[0]), 100, stream);
```

- to read a double precision floating-point value:

```
double fpoint;
fread((char *)&fpoint, sizeof(fpoint), 1, stream);
```

- to read an array of 50 floating-point values:

```
float fpoint[50];
fread((char *)fpoint, sizeof(fpoint[0]), 50, stream);
```

To get the equivalent **fwrite** calls, just substitute "fwrite" in place of "fread" in the previous examples. You can see how much more flexible **fread** and

`fwrite` are than `getw` and `putw`. Whereas `getw` and `putw` are limited to reading or writing a single four-byte integer per call, `fread` and `fwrite` can be made to read or write any number of variables of any type.

Stream Status and Control Routines

This section discusses standard input/output routines which enable you to

- determine whether or not an error has occurred on an open stream (`feof`, `ferror`, `clearerr`)
- re-position the location of the next input/output operation on an open stream (`rewind`, `ftell`, `fseek`)
- control various attributes of an open stream, such as buffering, flushing, etc. (`fclose`, `setbuf`, `fflush`, `freopen`)
- convert a file pointer to a file descriptor, and vice versa (`fileno`, `fdopen`)

Stream Status Inquiry Routines

This section describes three routines, `feof`, `ferror`, and `clearerr`, which enable you to determine the status of an open stream at any given time.

`feof` is a macro defined in `<stdio.h>` which returns a non-zero value if the end-of-file has been reached on an input stream. Its syntax is:

`feof(`*stream*`);`

Do you remember the example program which illustrated the use of `getw` and `putw`? It was noted that comparing `getw`'s return value to the constant EOF was faulty, because `getw` returns an integer, and EOF is defined to be a valid integer (-1). How then do you determine if end-of-file has been reached when routines like `getw` are being used? You use `feof`.

The example program for **getw** and **putw** can be changed to use **feof**:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int word;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }

   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
      fprintf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   datale = fopen("dfle", "w");
   if(datale == NULL) {
      fprintf("Can't create dfle file.\n");
      exit(1);
   }

   datagt = fopen("dfgt", "w");
   if(datagt == NULL) {
      fprintf("Can't create dfgt file.\n");
      exit(1);
   }

   for(;;) {
      if((word = getw(dfile)) != EOF) {
         if(word <= 32767 && word >= -32767)
            putw(word, datale);
         else
            putw(word, datagt);
```

```
        } else {
            if(feof(dfile))
                break;
            else
                putw(word, datale);
        }
    }

    exit(0);
}
```

An infinite loop is set up around the `getw/putw` process. Whenever `getw` returns an integer equal to EOF, `feof` is used to find out if end-of-file has been reached. If it has, the loop (and the program) terminates; if not, the integer is written on `dfle`, and the loop continues.

`ferror` is a routine which examines the specified stream to determine whether or not a read or write error has occurred. Its syntax is

`ferror(`*stream*`);`

`ferror`, like `feof`, is intended to clarify ambiguous return values from standard input/output routines. Actually, only `getw` and `putw` require the use of `ferror` to determine if an error has occurred. Both of these routines return EOF on end-of-file or error. Since these routines deal with integer data, however, you need `feof` and `ferror` to determine if the EOF returned actually indicated an error or an end-of-file, or if it's just a -1.

If an error has occurred on a stream, `ferror` returns a much non-zero value.

Whenever an error occurs on an open stream, a flag is set to indicate the error. It is this flag that `ferror` checks to determine whether or not an error has occurred. This flag is *not* reset when it is checked. Thus, if an error has occurred, the error flag for that stream remains set. This could lead to misleading information if an `ferror` call indicates that an error has occurred, when in reality the error occurred long ago. The `clearerr` routine clears (or resets) the error indication flag for the specified stream. This routine should be used whenever an error has been indicated, so that the same error is not indicated at a later time. `clearerr`'s syntax is:

`clearerr(`*stream*`);`

Because `ferror` and `clearerr` are used infrequently in typical programs, no examples are given specific to their use. The `feof` example above illustrates the general scenario in which all three of these routines are used.

Repositioning Stream Input/Output Operations

There are three routines, `rewind`, `ftell`, and `fseek`, which enable you to move the location of the next input/output operation on an open stream.

Its syntax is

```
rewind(stream);
```

For example, suppose a particular application program can put a password on a data file it uses. This password is stored in encrypted form on the first line of the file. The line is recognized as a password line if the first two characters are "*P". If the file has no password line, then access to the file is unrestricted. If a password line is found, the user is prompted for the password before access is permitted. The following code can be used to look for a password line:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   FILE *pswd;
   char line[256];

   if(argc != 2) {
      fprintf(stderr, "Usage: getpswd file\n");
      exit(1);
   }

   pswd = fopen(argv[1], "r");
   if(pswd == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
     }

   fgets(line, 256, pswd);
   if(line[0] == '*' && line[1] == 'P') {

   /* ask for and check password */

   } else
      rewind(pswd);

                . . ./* application program goes here */s

   exit(0);
}
```

If the first two characters of the first line are **P*, then code is executed which asks for and checks a password. However, if the first line is not a password line, the file is assumed to be unprotected, and the line just read is probably part of the data. Thus, the file must be rewound so the data contained in the first line is available to the application program.

The `ftell` routine returns a long integer specifying the current position of the next input/output operation on an open stream. This position is expressed as a byte offset relative to the beginning of the open file. Its syntax is as follows:

```
ftell(stream);
```

The `fseek` routine enables you to re-position the next input/output operation on an open stream to any location you wish. Its syntax is:

```
fseek(stream, offset, ptrname);
```

where *stream* is a file pointer to the open stream, *offset* is a long integer specifying the number of bytes to skip over, and *ptrname* is an integer indicating the reference point in the file from which *offset* bytes are measured. The possible values for *ptrname* are:

0 Move *offset* bytes from the beginning of the file.

1 Move *offset* bytes from the current position in the file.

2 Move *offset* bytes from the end of the file.

offset can be either negative or positive, indicating backward or forward movement in the file, respectively.

The following program illustrates the use of the `ftell` and `fseek` library routines. The program prints each line of an n-line file in this order: line 1, line n, line 2, line $n-1$, line 3, ... from the beginning of the file.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   char line[256];
   int newlines;
   long front, rear, ftell();
   FILE *fp;

   front = 0;
   rear = 0;

   if(argc < 2) {
      fprintf(stderr, "Usage:  print filename\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   newlines = countnl(fp) % 2;

   fseek(fp, 0, 2);
   rear = ftell(fp);

   while(front < rear) {
        fseek(fp, front, 0);
        fgets(line, 256, fp);
        fputs(line, stdout);
        front = ftell(fp);
        findnl(fp, rear);
        rear = ftell(fp);
        if(newlines == 1) {
             if(rear <= front)
```

```
                    break;
            }
            fgets(line, 256, fp);
            fputs(line, stdout);
        }

        exit(0);
}

countnl(fp)
FILE *fp;
{
        char c;
        int count = 0;

        while((c = getc(fp)) != EOF) {

if(c == '\n')

        count++;
        }
        rewind(fp);
        return(count);
}

findnl(fp, offset)
FILE *fp;
long offset;
{
        char c;

        fseek(fp, (offset-2), 0);
        while((c = getc(fp)) != '\n') {

fseek(fp, -2, 1);
        }
}
```

This program uses **ftell** and **fseek** to print lines from a file starting at the beginning and the end of the file, and converging toward the center. The **countnl** (count new-lines) function counts the number of lines in the file so the program can decide whether or not to print a line in the final loop (this prevents the middle line being printed twice in files with an odd number of lines). The **findnl** (find new-line) function seeks backwards in the file for the next new-line. When found, this positions the next input/output operation such that **fgets** gets the next line back from the end of the file.

Note the use of **fseek** in this program. All three types of seeks are represented here. The first **fseek** of the program is done relative to the end of the file. All other **fseek**s in the main program are done relative to the beginning of the file. Finally, **findnl** contains an **fseek** which is relative to the current position.

Recall the employee data routine, where each employee is described by the structure:

```
struct emp {
          char      name[40]; /* name */
          char      job[40];  /* job title */
          long      salary;   /* salary */
          char      hire[6];  /* hire date */
          char      curve[2]; /* pay curve */
          int       rank;     /* percentile ranking */
}
```

That routine simply read in the data for 400 employees all at once. Suppose you want the program to be selective, so that you can specify (by employee number, 1 through 400) *which* employee's information you want. This is easily done using **fseek**. The following program fragment shows how:

```
    :
int empno, bytes;
long total;
FILE *data;
struct emp empinfo;

/* check for usage error and open data file */
    :
sscanf(argv[1], "%d", &empno);
bytes = sizeof(empinfo);
total = (empno - 1) * bytes;
fseek(data, total, 0);
fread((char *)&empinfo, sizeof(empinfo), 1, data);

/* print out desired information */
    :
```

In this program, **argv[1]** contains, via a command-line argument, the employee number about whom information is desired. This employee number is converted to integer form using **sscanf**. The number of bytes per employee structure is obtained using **sizeof** and is stored in **bytes**. The total number of bytes to skip in the data file is found by multiplying the employee number (minus one) times the number of bytes per employee structure. This is stored in **total**. **fseek** is then used to seek past the specified number of bytes relative to the beginning of the data file. This leaves the next input/output operation positioned at the start of the specified employee's information. The information is read using **fread**.

Note If you have a stream which is open for both reading and writing, a read operation cannot be followed by a write operation without one of the following occurring first: a `rewind`, an `fseek`, or a read operation which encounters end-of-file. Similarly, a write operation cannot be followed by a read operation unless a `rewind` or `fseek` is performed.

Stream Control Routines

The routines described here help you control certain attributes of file pointers. The routines described are `setbuf`, `setvbuf`, `fflush`, and `freopen`.

setbuf

`setbuf` and `setvbuf` routines enable you to assign your own buffering to an open stream. `setbuf` syntax is:

```
setbuf(stream, buffer);
```

where *stream* is a file pointer to an already-open stream, and *buffer* is a pointer to a character array or is NULL.

Normally (without user intervention), a standard input/output buffer is obtained through a call to `malloc` (see *malloc*(3C)) upon the first call to `getc` or `putc` (which all input/output routines eventually call). The standard input/output system normally buffers input/output in a buffer which is `BUFSIZ` bytes long. Exceptions are `stdout`, which, when directed to a terminal, is line-buffered, and `stderr`, which is normally unbuffered.

setbuf enables you to change the buffer used for all standard input/output routines. For example, the following code fragment causes the array **buffer** to be used for buffering:

```
:

FILE *fp;
char buffer[BUFSIZ];

fp = fopen(argv[1], "r");
:

setbuf(fp, buffer);
:
```

This fragment shows the correct order of events. First, the file is opened (it need not be opened for reading), then the buffering is assigned using **setbuf**. From that point on, any input taken from

Buffering can be eliminated altogether by specifying the NULL pointer in place of the buffer name, as in

```
setbuf(fp, NULL);
```

This causes input or output using **fp** to be completely unbuffered.

setbuf is limited to buffer sizes of either **BUFSIZ** bytes or zero. **setbuf** assumes that the character array pointed to by "buffer"is **BUFSIZ** bytes. Passing **setbuf** a (non-NULL) pointer to a smaller array can cause severe problems during operation because the standard input/output routines may overwrite memory following the end of the too-small buffer.

Note

Using an *automatic* array as a standard input/output buffer can be dangerous. Automatic variables are only defined in the code block in which they are declared. Thus, buffering which relies on an automatic array is only in effect during the current code block (main program or function). If you pass a file pointer to another function, and the stream pointed to by that file pointer is buffered using an automatic array, then memory faults or other errors can occur. If you use an automatic array for stream buffering, the stream should be used and closed only in the code block containing the array declaration. To avoid this restriction, use *external* arrays for buffering:

```
extern char buffer[BUFSIZ];
:
setbuf(fp, buffer);
```

setvbuf

`setvbuf`, like `setbuf`, enables you to assign a character array for buffering, but also provides the means to specify the size of the buffer to be used and the type of buffering to be done. `setvbuf` syntax is:

`setvbuf(`*stream*`,` *buffer*`,` *type*`,` *size*`)`

where *stream* is a file pointer to an already-open stream, *buffer* is a pointer to a character array or is NULL, *type* tells how *stream* is to be buffered, and *size* defines how large the *buffer* is. Acceptable values for *type* (defined in `<stdio.h>`) include:

`_IOFBF`	Input/output is fully buffered.
`_IOLBF`	Output is line buffered. The buffer is flushed each time a new line is written, the buffer is full, or input is requested.
`_IONBF`	Input/output is completely unbuffered.

If type `_IONBF` is specified, *stream* is totally unbuffered. Since no buffer is needed, values for *buffer* and *size* are ignored. For example, the following two calls, though different, are functionally identical:

```
setvbuf(fp, NULL, _IONBF, 0)
setbuf(fp, NULL)
```

When *type* is `_IOFBF` or `_IOLBF`, buffering for *stream* is determined by *buffer* and *size*. If *buffer* is not the NULL pointer, it must point to a character array of *size* bytes. All buffering of *stream* is then handled through this array.

```
:
FILE *fp;
char buffer [256]
char *filename;
int ... retcode;
fp=fopen(filename, "w");
retcode=setvbuf(fp, buffer, _IOFBF, 256);
if (retcode !=0) error c);
```

This fragment buffers stream `fp` through a 2048-byte buffer that is allocated by the system.

fflush

The `fflush` routine forces all buffered data for an output stream to be written out to that file. Its syntax is:

```
fflush(stream);
```

where *stream* is a file pointer to an output stream.

`fflush` is performed automatically by `fclose` (and, therefore, by `exit`). Therefore, there is often no reason to call `fflush` explicitly. Situations do arise, however, where it is necessary to manually `fflush` a stream. For example, data written to a terminal is line-buffered by default, which means that the system waits for a new-line before writing the buffer onto the terminal screen. This is often satisfactory, but there are times when you want whatever has been written so far to be written to the screen without waiting for the new-line. In such situations, `fflush` must be used.

Another situation when explicit **fflushing** is necessary arises whenever you have written less than a buffer-full of data to a file, and you want the contents of that file processed by another function or by an HP-UX command. Since less than a buffer-full of data was written, the data is still in the buffer and the file is still empty. Performing an **fflush** causes the buffered data to be written out to the file, enabling other functions or commands to access the file's contents.

Yet another situation in which a program should call **fflush** explicitly is when it has opened a stream for both reading and writing (**r+**). When switching from writing to reading, the program should call **fflush** (or **fseek**) before reading.

freopen

The final routine in this section is **freopen**. As its name implies, **freopen** enables you to, in a single step, close a stream and then re-open it with a different type and/or file name. Its syntax is:

freopen(*filename*, *type*, *stream*);

where *filename* is a pointer to a character string specifying the name of the source or destination file for the newly-created stream. *type* is identical to that of **fopen** discussed earlier. *stream* is a file pointer to the old stream, which is closed and then re-opened. The name of the file pointer remains the same.

For example, the following program accepts lines of data from your terminal and writes them into a file. When only a new-line is typed from the terminal, the program quits reading data, and echoes the contents of the file to the terminal.

```
#include <stdio.h>
main()
{
  FILE *fp, *oldfp;
  char line[80], *fgets( );

  fp = fopen("datafile", "w");
  if(fp == NULL) {
     fprintf(stderr, "Can't create datafile.\n");
     exit(1);
  }

  fgets(line, 80, stdin);
  while(line[0] != "\n") {
     fputs(line, fp);
     fgets(line, 80, stdin);
  }

  oldfp = freopen("datafile", "r", fp);
  if(oldfp == NULL) {
     fprintf(stderr, "Can't re-open datafile.\n");
     exit(1);
  }

  while(fgets(line, 80, fp) != NULL)
     fputs(line, stdout);

  fclose(fp);
  exit(0);
}
```

9 Just like `fopen`, `freopen` returns a NULL pointer if an error occurs. If successful, `freopen` returns the value of the old file pointer.

`freopen` is commonly used to attach the names `stdin`, `stdout`, and `stderr` to other files so that the source or destination of these file pointers can be redirected. For example:

```
freopen("/usr/lib/data/datafile", "r", stdin);
```

attaches `stdin` to the data file `/usr/lib/data/datafile`. Other functions can now be called that read from `stdin`, with the result that their source of input has been redirected. Similarly,

```
freopen("/users/bill/archives/cal.a", "a", stdout);
```

attaches `stdout` to the indicated file, thus redirecting any future `stdout` data to that file.

Converting between File Pointers and File Descriptors

A file pointer is actually a pointer to a structure containing information about a stream. This information includes a pointer to the beginning of the buffer; a pointer to the current location in the buffer; a flag specifying whether the stream is open for reading, writing, or both; a count of the characters in the buffer; and an integer called a **file descriptor**.

System calls, such as `open` and `creat`, return a file descriptor when a file is opened. System calls use file descriptors to refer to open files in much the same way that library routines use file pointers. (The main difference between using a file descriptor and using a file pointer is that a file descriptor has no associated buffering.) Since a program often contains both system calls and library routines, a way of converting between file pointers and file descriptors is provided.

Note

Exercise caution when converting between file pointers and file descriptors. When converting a file pointer to a file descriptor, a program should call **fflush** first.

In general, never convert file pointers to file descriptors unless you need a file descriptor for a system call that provides a utility not available in the C library package (such as *dup*(2) or *fcntl*(2)). Similarly, file descriptors should never be converted to file pointers unless a file descriptor has been created by a system call which provides a utility not provided in the C library package and you want to assign system buffering to it.

Two routines, **fileno** and **fdopen**, provide a way to convert between the two types of parameters. **fileno** is a macro which, given a file pointer, returns the associated file descriptor. Its syntax is

```
fileno(stream);
```

where *stream* is a file pointer to an open stream whose associated file descriptor is desired. Thus,

```
.
.
.
FILE *fp;
.
.
.
int fd;
fp = fopen("file1", "r");
fd = fileno(fp);
```

returns the integer file descriptor in **fd**, associated with the file pointer **fp**.

The **fdopen** routine enables you to convert a file descriptor into a file pointer. Its syntax is:

fdopen(*fildes*, *type*);

where *fildes* is an integer file descriptor obtained from the **open**, **dup**, **creat**, or **pipe** system calls. *type* is the same as that for **fopen** discussed earlier. Thus,

```
:
int fd;
FILE *fp;
:/* obtain fd via appropriate system call */
:
fp = fdopen(fd, "r");
if(fp == NULL) {
     fprintf(stderr, "Can't convert file descriptor.\n");
     exit(1);
}
 :
```

converts the file descriptor **fd** into a file pointer, **fp**. **fdopen** returns a NULL pointer if the operation fails.

fdopen can be useful for opening a file in a way unlike any of the standard types of **fopen**.

```
:
# include <fcntl.h>
:
int fd;
FILE *fp
char *filename;
:
fd= open(filename, O_WRONLY|O_CREAT, 0666);
fp= fdopen(fd,"w");
fseek(fd,0L,2)
```

This code fragment uses the **open** system call to open a file for general write access, then uses **fdopen** to assign buffering to the file. The constants **O_WRONLY** and **O_CREAT** are defined in the include file **/usr/include/fcntl.h**, and are described in *open*(2). (**O_WRONLY** causes **open** to open the file for

writing only; `O_CREAT` creates the file if it does not already exist.) This technique opens the file in a way that does not correspond exactly to any of the available types in `fopen`: "w" would truncate the current file contents, "r+" would fail if the file does not already exist (and would allow reading of the file), and "a" does not permit seeking backwards and rewriting the current file contents.

Inter-Process Communication

So far, you've been communicating between an active process (your program) and a passive object (a file). What if you want to communicate between two active processes? Suppose you want to create a stream between two programs, with one program (process) pumping data onto the stream, and the other reading data from the other end. How is this done? The `popen` routine exists for this purpose. Its syntax is:

`popen(`*command*`,` *type*`);`

where *command* is a pointer to a character string specifying a command line. *type* is a pointer to a single-character string which is either `"r"` (for reading) or `"w"` (for writing).

For example, suppose you are writing a program that processes text in some way. Your program handles normal text perfectly, but unfortunately your source files are all coded in `troff` constructs. If you could filter out all the `troff` constructs, your program would work fine. This can be done using pipes and the HP-UX command called `deroff`, which filters out `troff` constructs. All you have to do is make sure that all input to your program passes through `deroff` first. Here's how:

```
#include <stdio.h>
main()
{
   FILE *popen(), *fp;

   fp = popen("deroff /users/bin/text/*.tx", "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't create stream.\n");
      exit(1);
   }

   /* begin processing text; read text from fp! */
   :
   pclose(fp);
```

popen returns a file pointer to the newly-opened stream. If an error occurs, a NULL pointer is returned. When successfully executed, **popen** enables your program to read from the file pointer **fp**, the data from which is the standard output from the **deroff** command. In this example, **deroff** is invoked such that it processes all files in **/users/bin/text** which end with **.tx**. Note that **popen**'s return value must be declared explicitly because it is not declared in **<stdio.h>**.

Because **deroff** processes **stdin** if no arguments are given, the following **popen** call enables your program to receive filtered text from **stdin** instead of from ordinary files:

```
fp = popen("deroff", "r");
```

The result of executing the previous example is exactly the same as if you had typed

deroff /users/bin/text/*.tx | *yourprogram*

at your keyboard in response to a shell prompt.

Streams that are opened by **popen** must be closed with **pclose**. Thus, the following call closes the stream created in the previous example:

```
pclose(fp);
```

If a *type* of `w` is specified instead of `r`, then the data flow is reversed, with the result that your program supplies the data for the specified *command*.

Note that, though `popen`'s return value is called a file pointer, it is actually somewhat different than the file pointers you are already familiar with. In general, a file pointer returned by `popen` should not be used in those previously-discussed library routines which modify file pointers returned by `fopen`. Also, file pointers opened by `popen` must be closed with `pclose`; `fclose` is not sufficient.

So far, `popen` has been characterized as a "filter-maker", in that streams to or from a command have been created so that data can be modified in some way before being passed on. Sometimes, however, `popen` is used to execute a command which supplies information valuable to the program. For example, the `find` command accepts dot (.) as a valid directory name. Upon receipt of a dot, `find` discovers the actual path name of dot by creating a stream from the `pwd` command, as follows:

```
char dir[100];
FILE *popen(), *fp;

fp = popen("pwd", "r");
if(fp == NULL) {
     fprintf(stderr, "Can't execute pwd.\n");
     exit(1);
}
fgets(dir, 100, fp);
:
pclose(fp);
```

The preceding example reads the output of the `pwd` command into the character array `dir`, thus supplying the current value of dot. The following program creates a list of the login names of users currently logged in:

```
#include <stdio.h>
main()
{
   char name[10], line[80], *fgets();
   FILE *popen(), *fp;

   fp = popen("who", "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't execute who.\n");
      exit(1);
   }

   printf("Users currently logged in:\n");
   while(fgets(line, 80, fp) != NULL) {
      sscanf(line, "%s", name);
      printf("\t%s\n", name);
   }

   pclose(fp);
   exit(0);
}
```

A stream is created for reading from the `who` command. Each line from `who` is read, and the first field from each line is read and printed.

You can have only one `popen`-ed stream in a process at any given time.

10

Standard Character, String, and Date Manipulation Routines

This chapter describes standard `libc` routines that

- convert character case
- classify characters
- manipulate strings
- perform date and time manipulation

Converting between Uppercase and Lowercase

Four routines are documented under *conv*(3C) which enable you to convert between upper- and lowercase. They are `toupper`, `tolower`, `_toupper`, and `_tolower`.

`toupper` and `tolower` are functions which accept a single integer argument in the range −1 through 255. If the integer taken as a character represents a lower-case character, `toupper` returns the corresponding upper-case character. Similarly, `tolower` returns the corresponding lower-case character. Both routines return the argument unchanged if it does not represent a lower-case character (`toupper`) or an upper-case character (`tolower`).

`_toupper` and `_tolower` are macros defined in `<ctype.h>`. `_toupper` accepts a single character argument and returns the corresponding upper-case character. Similarly, `_tolower` returns the corresponding lower-case character for its argument. If a character is specified that is not a lower-case character (`_toupper`) or an upper-case character (`_tolower`), the macros simply return that character. Negative values, however, will not convert properly.

The macro versions of these routines are faster than the functions. The function versions are useful when you need to pass such a function to another routine.

Character Classification

The *ctype*(3C) entry in the *HP-UX Reference* lists routines which test their single argument and return a non-zero value if the test is positive, and 0 otherwise.

All of these routines are macros defined in `<ctype.h>`. Because the syntax for all `ctype` macros is identical, the following example can easily be be modified for all `ctype` macros:

```
:
for(i=0; array[i] != NULL; i++) {
     if(islower(array[i]))
          array[i] = _toupper(array[i]);
}
:
```

This program fragment shows one way to change all occurrences of a lowercase character in `array` to uppercase using the macro `_toupper`. The call to the `islower` macro ensures that only lowercase characters are passed to `_toupper`.

String Manipulation Routines

String(3C) in the HP-UX Reference manual documents an extensive list of string manipulation routines enabling you to perform several operations on character strings. This section describes the *string*(3C) package in detail.

Concatenating Strings

`strcat` and `strncat` enable you to append a copy of one string onto the end of another. Their syntax is:

```
strcat(s1, s2);

strncat(s1, s2, n);
```

where *s1* and *s2* are character pointers to NULL-terminated character strings. `strcat` appends the entire string pointed to by *s2* (up to the first NULL character encountered) on the end of string *s1*. `strncat` does the same thing, except that at most *n* characters are appended to *s1* (or up to a NULL character, whichever comes first). (Note that string *s2* need not be NULL-terminated when using `strncat` if *n* is less than or equal to the length of *s2*.) Both routines return a character pointer to the NULL-terminated result.

Neither of these routines checks to make sure that there is room in *s1* for the additional characters of *s2*. Thus, to be safe, *s1* should *always* be a declared array having plenty of space for the additional characters of *s2*, plus a terminating NULL character.

Copying Strings

`strcpy` and `strncpy` copy one string of characters into another. Their syntax is:

```
strcpy(s1, s2);

strcpy(s1, s2);

strncpy(s1, s2, n);
```

where *s2* is a character pointer to the string to be copied, and *s1* is a character pointer to the beginning of the string into which the contents of string *s1* are copied. `strcpy` copies the entire string, up to (and including) the first NULL encountered. `strncpy` copies up to *n* characters, or up to (and including) the first encountered NULL, whichever occurs first. (String *s2* need not be

NULL-terminated when using **strncpy** if *n* is less than or equal to the length of *s2*.) Both routines return the value of *s1*.

The following program uses the **strcat** routine discussed earlier and **strcpy** to build a character string representing the lowercase alphabet, one character at a time.

```
#include <stdio.h>
main()
{
   int b = 'b', z = 'z', i;
   char alpha[30], chr[4];

   chr[1] = NULL;
   strcpy(alpha, "a");
   printf("%s\n", alpha);

   for(i = b; i <= z; i++) {
        chr[0] = i;
        strcat(alpha, chr);
        printf("%s\n", alpha);
   }
}
```

The array **chr** is always going to be a two-character array consisting of the next character in the alphabet followed by NULL. Thus, the second element of **chr** is set to NULL early in the program. The first **chr** element is then successively set to the next lowercase character in the **for** loop, and the resulting two-character string is concatenated onto the end of the alphabet assembled so far in **alpha**. Note the use of **strcpy** to initialize **alpha**. Remember that C transforms one or more characters enclosed in double quotes into a character pointer to those characters followed by a NULL. Thus, the **strcpy** statement above copies the character "a" followed by a NULL character into **alpha**.

There are some things to be aware of when using **strcat**, **strncat**, **strcpy**, and **strncpy**. These routines all modify string *s1* in some way, but none of them check for into overflow in that string. Therefore, be sure there is enough room in *s1* to hold the added or copied characters plus at| terminating NULL. Also, be sure you use a character *array* for *s1* (*not* just a character

pointer), especially when using `strcat` or `strncat`. This is because an explicitly-declared array has sufficient memory allocated to it to contain all of its elements, but a character pointer simply points to a single location in memory. Concatenating a string to the end of a string contained in an array is guaranteed to work, provided the array is large enough. However, concatenating a string to a string of characters referenced by a simple character pointer is dangerous, since the concatenated characters could overwrite data in memory. For example,

```
char array[100], *ptr = "abcdef";
⋮
strcat(array, ptr);
```

works fine, since you are guaranteed that 100 storage elements have been set aside for the array. However,

```
char *ptr1 = "abcdef", *ptr2 = "ghijkl";
⋮
strcat(ptr1, ptr2);
```

is asking for trouble. Although C makes sure that there is enough room for the initializing strings ("abcdef" and "ghijkl" in this example), there are no guarantees that there is enough room to add characters to the end of one of these strings. Therefore, the last fragment could easily overwrite valid data occurring after the string pointed to by `ptr1`.

Comparing Strings

`strcmp` and `strncmp` compare two strings and return an integer indicating the result of the comparison. Their syntax is:

```
strcmp(s1, s2);

strncmp(s1, s2, n);
```

where *s1* and *s2* are character pointers to the. NULL-terminated character strings to be compared. `strcmp` compares the entire strings, stopping as soon as the result is determined. `strncmp` compares at most *n* characters of, both strings (neither string need be NULL-terminated if *n* is less than or equal to the length of the shorter string). The integer returned uses the following convention:

<0 *s1* is lexicographically less than *s2*.

=0 *s1* and *s2* are equal.

>0 *s1* is lexicographically greater than *s2*.

The following program fragment uses **strncmp** to analyze the contents of a file coded with the man macros (see *man*(7)). It reads each line of the file and keeps a count of the number of times selected macros are used, and prints a summary of its findings at the end.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    char *fgets(), line[100];
    FILE *fp;
    int nsh, npp, ntp, nrs, nre, npd, nip, nmisc, nlines;

    nsh = npp = ntp = nrs = nre = npd = nip = nmisc = nlines = 0;

    if(argc != 2) {
        fprintf(stderr, "Usage: count file\n");
        exit(2);
    }

    fp = fopen(argv[1], "r");
    if(fp == NULL) {
        fprintf(stderr, "Can't open %s.\n", argv[1]);
        exit(1);
    }

    while(fgets(line, 100, fp) != NULL) {
        if(strncmp(line, ".SH", 3) == 0)
            nsh++;
        else if(strncmp(line, ".PP", 3) == 0)
            npp++;
        else if(strncmp(line, ".TP", 3) == 0)
```

```
            ntp++;
        else if(strncmp(line, ".RS", 3) == 0)
            nrs++;
        else if(strncmp(line, ".RE", 3) == 0)
            nre++;
        else if(strncmp(line, ".PD", 3) == 0)
            npd++;
        else if(strncmp(line, ".IP", 3) == 0)
            nip++;
        else if(line[0] == '.')
            nmisc++;
        nlines++;
    }

    printf("No. of lines: %d\n\n", nlines);
    printf("No. of .SH's: %d\n", nsh);
    printf("No. of .PP's: %d\n", npp);
    printf("No. of .TP's: %d\n", ntp);
    printf("No. of .RS's: %d\n", nrs);
    printf("No. of .RE's: %d\n", nre);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .PD's: %d\n", npd);
    printf("No. of .IP's: %d\n", nip);
    printf("No. of misc. macros: %d\n", nmisc);

    fclose(fp);
    exit(0);
}
```

In the above program, **strncmp** is used to compare the first three characters of each line read. If the first three characters match a particular macro, the appropriate counter is incremented. If the line begins with ".", but is not one of the macros being searched for, the "miscellaneous" counter is incremented. The total number of lines in the file is also given.

Finding the Length of a String

The `strlen` routine returns an integer specifying the number of non-NULL characters in a string. Its syntax is:

```
strlen(s);
```

where `s` is a character pointer to the NULL-terminated string whose length is to be taken. For example, if you execute

```
len = strlen( string );
```

then the integer `len` contains the total number of non-NULL characters in the string pointed to by *string*. Thus,

```
string[len]
```

points to the terminating NULL in `string`.

Finding Characters in Strings

The `strchr`, `strrchr`, and `strpbrk` routines enable you to locate a particular character within a string.

`strchr` and `strrchr` return a character pointer to an occurrence of a specified character in a string. Their syntax is:

```
strchr(s, c);

strrchr(s, c);
```

where s is a character pointer to the string of interest, and c is a variable of type `char` specifying the character to search for.

`strchr` returns a character pointer to the first occurrence of character c in string s. Similarly, `strrchr` returns a character pointer to the last occurrence in string s. Both routines return a NULL if the character does not occur in the string pointed to by s. For example,

```
char *ptr, *strchr(), string[100];
⋮
while((ptr = strchr(string, '@') != NULL)
   *ptr = '#';
```

replaces all occurrences of @ in the array **string** with **#**, starting from the beginning of the array and working toward the end. The same operation can be done using

```
while((ptr = strrchr(string, '@')) != NULL)
   *ptr = '#';
```

which replaces all @'s with #'s, starting from the end of the array, working backward toward the beginning.

The **strpbrk** routine returns a character pointer to the first occurrence in string *s1* of any character contained in string *s2*, or NULL if none of the characters in *s2* occur in *s1*. Its syntax is:

```
strpbrk(s1, s2);
```

For example, suppose you have to read lines of input in which are embedded numerical data which must be read. For simplicity, assume that the following conventions are used:

- Positive numbers do not begin with **+**.
- Fractional numbers always begin with zero, as in **0.25**.
- The first occurrence of a digit in the string signals the beginning of the number to be read.

Given these rules, the following code fragment does the job:

```
char line[100], *chrs = "-0123456789", *ptr;
float value;
⋮
sscanf(ptr, "%f", &value);
⋮
```

The character pointer **chrs** is initialized to point to a string of characters which might introduce the embedded number. **strpbrk** then finds the first occurrence of one of these characters in **line**, and returns a pointer to that location in **ptr**. Finally, **ptr** is passed to **sscanf**, which interprets **ptr** as if it were a pointer to the beginning of a string from which input is to be taken. The number is read correctly because **ptr** points to the beginning of a number, and because the **%f** conversion terminates at the first inappropriate character.

Finding Characters Common to Two Strings

The `strspn` and `strcspn` routines return an integer giving the length of the initial segment of string *s1* which consists entirely of characters found in string *s2*. `strcspn` is similar, but returns an integer giving the length of the initial segment of *s1* which consists entirely of characters *not* found in string *s2*. Their syntax is:

```
strspn(s1, s2);

strcspn(s1, s2);
```

For example, suppose you have the following two strings:

```
"A tattle-tale never wins."
```

for string `s1`, and

```
" -Aatle"
```

for `s2`. Executing

```
strspn(s1, s2);
```

with the strings shown returns a value of 14, since the first 14 characters (`A tattle-tale`) in `s1` all occur in `s2`.

```
strcspn(s1, s2);
```

using the same strings, you get 0, because there is no initial segment of `s1` which contains characters not found in `s2`.

Breaking a String into Tokens

A **token** is a string of characters delimited by one or more token delimiters. The `strtok` routine divides string *s1* into one or more tokens. The token separators consist of any characters contained in string *s2*. Its syntax is:

```
strtok(s1, s2);
```

where *s1* is a character pointer to the string which is to be broken up into tokens, and *s2* is a character pointer to a string consisting of those characters which are to be treated as token separators.

strtok returns the next token from *s1* each time it is called. The first time **strtok** is called, both *s1* and *s2* must be specified. On subsequent calls, however, *s1* need not be specified (a NULL is specified in its place). **strtok** remembers the string from call to call. String *s2* must be specified each call, but need not contain the same characters (token separators) each time.

strtok returns a pointer to the beginning of the next token, and writes a NULL character into *s1* immediately following the end of the returned token. **strtok** returns a NULL when no tokens remain.

For example, suppose you are reading lines from **/etc/gettydefs**, which is the speed table for *getty*(1M). The lines in this file contain several fields delimited by hash mark characters (#). Thus, the following code could be used to read the fields of each line:

```
int count = 0;
char *delims = "#", *token, *arg1, *strtok(), line[256];

   arg1 = line;
   :
   while((token = strtok(arg1, delims) != NULL) {
      count++;
      printf("field %d: %s\n", count, token);
      if(count == 1)
         arg1 = NULL;
   }
```

This code makes sure that **strtok**'s first argument is NULL after the first call. Also, note that **delims** did not change from call to call, but it could have. This greatly increases the power of **strtok**, since it enables you to change the token delimiters between calls.

Date and Time Manipulation

ctime(3C) describes a set of routines which enable you to access the date and time as maintained by the system clock. This package knows about daylight saving time, and automatically converts between standard time and daylight saving time when appropriate. These routines are part of `libc`.

Most of the `ctime` routines require the quantity returned by the `time` system call (see *time*(2)), which is the number of seconds that have elapsed since 00:00:00 GMT (Greenwich Mean Time), January 1, 1970.

The `ctime` routine converts the *time*(2) value into a 26-character ASCII string of the form

```
Fri May 11 09:53:03 1984\n\0
```

where `\n` is a new-line character, and `\0` is a terminating NULL character. `ctime`'s syntax is:

```
ctime(value);
```

where *value* is a pointer to a long integer value representing the number of elapsed seconds since 00:00:00 GMT, January 1, 1970 (as returned by *time*(2)). Note that *value* is a pointer to the quantity returned by *time*(2), not just the quantity itself. Using *time*(2) and `ctime`, you can write your own simplified version of the `date` command:

```
#include <stdio.h>
main()
{
   char *str, *ctime();
   long time(), nseconds;

   nseconds = time((long *)0);
   str = ctime(&nseconds);
   printf("%s", str);
}
```

The rest of the routines in *ctime*(3C) require the include file `<time.h>`, which contains the definition of a structure called `tm`. This structure is made up of

several variables which contain the various components of the date and time. It looks as follows:

```
struct tm {
        int  tm_sec;
        int  tm_min;
        int  tm_hour;
        int  tm_mday;
        int  tm_mon;
        int  tm_year;
        int  tm_wday;
        int  tm_yday;
        int  tm_isdst;
};
```

The meaning associated with each structure member is:

`tm_sec`	The "seconds" portion of the system's 24-hour clock time.
`tm_min`	The "minutes" portion of the system's 24-hour clock time.
`tm_hour`	The "hours" portion of the system's 24-hour clock time.
`tm_mday`	The day of the month, in the range 1 through 31.
`tm_mon`	; The month of the year, in the range 0 through 11 (0 = January).
`tm_year`	The current year − 1900.
`tm_wday`	The day of the week, in the range 0 through 6 (0 = Sunday).
`tm_yday`	The day of the year, in the range 0 through 365.
`tm_isdst`	A flag which is non-zero if daylight saving time is in effect.

The `localtime` and `gmtime` routines accept a pointer to a quantity such as returned by *time*(2), and fill in the various components of the `tm` structure. `localtime` corrects the time for the local time zone and possible daylight saving time, while `gmtime` converts directly to GMT time (this is the time used by HP-UX). Both routines return a pointer to a structure of type `tm` which can be used to access the various components of the `tm` structure.

For example, the following code fragment assigns values to the `tm` structure members for the local time zone:

```
#include <time.h>
:
struct tm *ptr, *localtime();
long time(), nseconds;
:
nseconds = time((long *)0);
ptr = localtime(&nseconds);
```

Once this code is executed, you can use `ptr` to access the different components of the local time. For example, `ptr->tm_mon` references the month of the year, and `ptr->tm_wday` references the day of the week. (`gmtime` is used in exactly the same way, so this example suffices for it also).

The `asctime` routine converts the time contained in a `tm` structure into ASCII representation such as that returned by *date*(1) and `ctime`. Its syntax is:

`asctime(`*ptr*`);`

where *ptr* is a pointer to a structure of type `tm` whose members have previously been assigned values with `localtime` or `gmtime`, or explicitly by you. `asctime` returns a character pointer to the same NULL-terminated 26-character string as returned by `ctime`.

`asctime` provides a way for you to obtain the current time, modify it explicitly in some way, and then print the result in ASCII form. The `date` command shown earlier can be re-written using `localtime` and `asctime`:

```
#include <stdio.h>
#include <time.h>
main()
{
   long time(), nseconds;
   struct tm *ptr, *localtime();
   char *string, *asctime();

   nseconds = time((long *)0);
   ptr = localtime(&nseconds);

/* the user can modify the current time in tm here */

   string = asctime(ptr);
   printf("%s", string);
}
```

This program illustrates a rather indirect way to obtain the date, but it does enable you to modify the date stored in `tm` before you print it out. If all you want to do is print the date, the quickest way is to use the `time`/`ctime` combination.

Of all the `ctime` routines, perhaps the most useful is `localtime`. It enables you to break the current time up into chunks which can then be examined for such applications as personal calendar programs, program schedulers, etc. Many of the `tm` values can be used as indices into arrays containing strings identifying months and days. For example, declaring an external array like

```
char *month[] = { "January", "February", "March", "April",
                  "May", "June", "July", "August", "September",
                  "October", "November", "December"
                };
```

enables you to use `tm_mon` as an index into this array to obtain the actual month name. The same thing can be done with `tm_wday` if you initialize an array containing the names of the days of the week. The *ctime*(3C) package makes it easy to design programs that depend upon the time or date. Try creating your own versions of *calendar*(1), *at*(1), or even *cron*(1M)!

11

Standard Math Routines

This chapter describes standard math library routines found in the SVID math library `libm`, the POSIX math library `libM`, and the standard library `libc`. The math functions do such things as

- calculate absolute value
- exponentiation
- square roots
- logarithms
- trigonometric functions
- random number generation

Note For details on floating-point concepts, refer to Series 700/800 *HP-UX Floating-Point Guide*. That book provides detail on such topics as the IEEE floating-point standard, exception handling, and math libraries.

The math.h Header File

To use math routines, a program should usually `#include` the header file `<math.h>`. This file contains type declarations of all the math routines that do not return an `int`, and a definition of the constant `HUGE`. Many math routines return a "huge" value when an error occurs, so `HUGE` is set equal to this "huge" value, enabling a program to check for errors easily.

The Math Libraries

Some of the math routines reside in the standard C library, `libc`, but many reside in the SVID math library, `libm`, and the POSIX math library `libM`. Therefore, when writing programs that use the routines described here, be sure to link a program with a math library. For example, to compile a C program named `mprog.c` that calls math routines, you could use:

```
$ cc mprog.c -lm
```

If your program must be ANSI-compliant, be sure to compile in ANSI mode and to use the POSIX math library:

```
$ cc -Aa mprog.c -lM
```

To determine precisely which library contains a particular math routine, refer to the *HP-UX Reference* page that describes the routine.

Note

On Series 700/800 systems, faster and more precise versions of the math libraries reside in the directories `/lib/pa1.1` and `/usr/lib/pa1.1`. To link with these libraries, use the `+DA` option, as described in the section "Selecting Faster Libraries" in Chapter 2. These libraries also contain many useful non-standard functions in addition to the standard ones described in this chapter (see the *HP-UX Floating-Point Guide* for details).

By default, Series 700 compilers automatically link with the faster PA1.1 libraries because exceptional performance is the primary concern of Series 700 applications. Since compatibility is usually the goal of Series 800 applications, Series 800 compilers link with the slower PA1.0 libraries by default.

Absolute Value Functions

The `abs` (*abs*(3C)) and `fabs` functions (see *floor*(3M)) return the absolute value of their integer or floating-point argument, respectively. For example, the following program calculates integer absolute values until a zero is entered from the keyboard:

```
main()
{
   int value;

   printf("Enter value: ");
   scanf("%d", &value);
   while(value != 0) {
     printf("Absolute value of %d is %d.\n", value, abs(value));
     printf("Enter value: ");
     scanf("%d", &value);
   }
   exit(0);
}
```

The floating-point equivalent of the previous program is shown below:

```
main()
{
 double value, fabs();

 printf("Enter value: ");
 scanf("%lf", &value);
 while(value != 0.0) {
  printf("Absolute value of %.12g is %.12g.\n",value,fabs(value));
  printf("Enter value: ");
  scanf("%lf", &value);
 }
 exit(0);
}
```

Power, Square Root, and Logarithmic Functions

This section describes the following five functions, all of which are found under *exp*(3M) in the *HP-UX Reference*:

`exp(`x`)`	Returns e to the x power.
`log(`x`)`	Returns the natural logarithm of x ($\ln(x)$).
`log10(`x`)`	Returns the common logarithm of x ($\log(x)$).
`pow(`x`, `y`)`	Returns x to the y power.
`sqrt(`x`)`	Returns the square root of x.

All functions return `double` values, and expect `double` arguments. Since their syntax is similar, the following logarithm calculator example shows all five of these functions:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Natural logarithm of %.12g = %.12g\n",
          value, log(value));
   printf("Common logarithm of %.12g = %.12g\n",
          value, log10(value));
}
```

This program accepts its single argument, and returns the natural and common logarithms of that argument.

Trigonometric Functions

A full set of trigonometric functions are provided in the math library. They are as follows:

`sin(`x`)`	Returns the sine of the radian argument x.
`cos(`x`)`	Returns the cosine of the radian argument x.
`tan(`x`)`	Returns the tangent of the radian argument x.
`asin(`x`)`	Returns the arc sine of x in the range -pi/2 to pi/2, where $-1 <= x <= 1$.
`acos(`x`)`	Returns the arc cosine of $x$in the range 0 to pi, where open $-1 <= x <= 1$.
`atan(`x`)`	Returns the arc tangent of $x$in the range -pi/2 to pi/2.
`atan2(`y`, `x`)`	Returns the arc tangent of yxin the range -pi to pi.
`sinh(`x`)`	Returns the hyperbolic sine of the radian argument x.
`cosh(`x`)`	Returns the hyperbolic cosine of the radian argument x.
`tanh(`x`)`	Returns the hyperbolic tangent of x.

Figure 11-1 shows a program that uses some of these routines, as well as two routines from the previous section, to obtain the dimensions and angles of a right triangle:

```
#include <stdio.h>
#include <math.h>
main()
{
   double sideA, sideB, sideC, anga, angb, tempC;
   double pi = fabs(acos(-1.));
   double torads = pi/180.;
   double todegs = 180./pi;
   double angc = 90.;

   printf("Using the following conventions for sides and angles:\n");
   triangle();
   printf("\nEnter all known information:\n");
   printf("\tA = ");
```

```
scanf("%lf", &sideA);
printf("\tB = ");
scanf("%lf", &sideB);
printf("\tC = ");
scanf("%lf", &sideC);
printf("\tAngle a = ");
scanf("%lf", &anga);
printf("\tAngle b = ");
scanf("%lf", &angb);
if(sideA && sideB && sideC) {
     tempC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
     if(fabs(sideC - tempC) > 0.001) {
          printf("Sides invalid.\n");
          exit(1);
     }
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA && sideB) {
     sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideB && sideC) {
     sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA && sideC) {
     sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
     anga = acos(sideB/sideC) * todegs;
     angb = 90. - anga;
} else if(sideA) {
     if(anga && angb) {
          sideC = sideA/cos(angb*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
     } else if(anga) {
          sideC = sideA/sin(anga*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
          angb = 90. - anga;
     } else if(angb) {
          sideC = sideA/cos(angb*torads);
          sideB = sqrt(pow(sideC, 2.) - pow(sideA, 2.));
          anga = 90. - angb;
```

```
        } else {
            printf("Insufficient information.\n");
            exit(1);
        }
    } else if(sideB) {
        if(anga && angb) {
            sideC = sideB/sin(angb*torads);
            sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
        } else if(anga) {
            sideC = sideB/cos(anga*torads);
            sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
            angb = 90. - anga;
        } else if(angb) {
            sideC = sideB/sin(angb*torads);
            sideA = sqrt(pow(sideC, 2.) - pow(sideB, 2.));
            anga = 90. - angb;
        } else {
            printf("Insufficient information.\n");
            exit(1);
        }
    } else if(sideC) {
        if(anga && angb) {
            sideA = sideC * cos(angb*torads);
            sideB = sideC * sin(angb*torads);
        } else if(anga) {
            sideA = sideC * sin(anga*torads);
            sideB = sideC * cos(anga*torads);
            angb = 90. - anga;
        } else if(angb) {
            sideA = sideC * cos(angb*torads);
            sideB = sideC * sin(angb*torads);
            anga = 90. - angb;
        } else {
            printf("Insufficient information.\n");
            exit(1);
        }
    } else {
        printf("Insufficient information.\n");
        exit(1);
    }
```

```
    printf("\n\tSide A = %.2f\t\tAngle a = %.2f degrees\n", sideA, anga);
    printf("\tSide B = %.2f\t\tAngle b = %.2f degrees\n", sideB, angb);
    printf("\tSide C = %.2f\n", sideC);
}
triangle()
{
    FILE *fopen(), *tri;
    char line[50], *fgets();

    tri = fopen("triangle", "r");
    if(tri == NULL) {
        printf("Cannot open triangle file.\n");
        exit(1);
    }

    while(fgets(line, 50, tri) != NULL)
        fputs(line, stdout);
    fclose(tri);
}
```

Figure 11-1. triangle.c—Get Dimensions of Right Triangle

The `triangle` function prints out the contents of a file in the current directory called `triangle`. The contents of this file should contain an ASCII approximation of a right triangle:

```
                /|
               / |
              /  |
             / a |
            /    |
       C /       | B
        /        |
       /         |
      /          |
     / b     c  _|
    /_________|_|
          A
```

This triangle made up of slashes, vertical bars, and underscores, shows the naming convention for the sides and angles. The program then asks for the

known data; enter a value of zero for those parameters that are unknown. The dimensions and angles are then calculated based on the data you have supplied. If there is insufficient information, you are told about it.

The hyperbolic functions are found under *sinh*(3M) in the *HP-UX Reference*.

Calculating Upper and Lower Bounds

Two functions, `floor` and `ceil` (see *floor*(3M)), enable you to obtain integers (returned as `doubles`) defining an upper and a lower bound for a number or a series of numbers. `floor` returns a double precision representation of the the largest integer which is still not greater than `floor`'s argument. Similarly, `ceil` returns a double precision representation of the smallest integer which is still greater than `ceil`'s argument.

The following program returns the floor and ceiling values for the number specified as its argument:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   double value;

   sscanf(argv[1], "%lf", &value);
   printf("Floor = %g; Ceiling = %g\n", floor(value), ceil(value));
}
```

If you type this in and run it, you see that `floor` and `ceil` provide two `double` values representing the smallest range in which the numbers used to obtain that range will fit. For example, if you have a program which reads three values from a source file, and these values are 4.79, 19.6, and 21.1, you can get the smallest possible range in which these numbers fit by running `floor` on each number (and keeping the smallest floor value), and then running `ceil` on each number (and keeping the largest ceiling value). For the above three numbers, this yields a floor value of 4, and a ceiling value of 22.

Calculating Remainders

This section covers two functions, `fmod` and `modf`. The `fmod` function (see *floor*(3M)) returns the remainder (in double precision form) resulting from dividing `fmod`'s first argument by its second. For example,

```
fmod(10., 4.)
```

divides 10 by 4, and returns the remainder (2.0, in this case). The following program accepts two numbers, divides the first by the second, and displays the results in a form showing the number of times the divisor goes evenly into the dividend, and the remainder, if any:

```
#include <math.h>
main(argc, argv)
int argc;
char *argv[];
{
   int result;
   double number, div, rem;

   sscanf(argv[1], "%lf", &number);
   sscanf(argv[3], "%lf", &div);

   result = number/div;
   printf("%g = (%d)(%g)", number, result, div);
   if((rem = fmod(number, div)) != 0.0)

printf(" + %g\n", rem);
}
```

This program is set up so that it can be invoked in sentence style. If you name the compiled version of this program "divide", then you can say

```
$ divide 33.27 by 11
```

Since argv[2] is ignored in the code, `by` is harmless, and the two numbers are parsed correctly.

The other function, **modf** (see *frexp*(3C)), is not really a remainder function in the same sense that **fmod** is a remainder function. In **fmod**, a division actually takes place. In **modf**, however, no division takes place. **modf** simply accepts a **double** value, and splits it into its integer and fractional parts. Syntax is:

modf(*value*, *iptr*);

where *value* is the number to be split into two parts, and *iptr* is a pointer to a **double** variable where the integer part of *value* is to be stored. **modf**'s return value is the signed fractional part of *value*.

The following program shows a way to use **modf**:

```
main(argc, argv)
int argc;
char *argv[];
{
    double value, iptr, frac, modf();

    sscanf(argv[1], "%lf", &value);
    frac = modf(value, &iptr);
    printf("Integer part: %g; Fractional part: %g\n", iptr, frac);
}
```

The program accepts one argument, the value, and then prints the integer and fractional parts of that value. Note that the address of **iptr** is passed to **modf**, because **modf** expects the address of a **double** variable where the integer part can be stored.

Calculating A Hypotenuse

The **hypot** function (see *hypot*(3M)) returns the square root of the sum of the squares of its two arguments, yielding the length of the hypotenuse of a right triangle, or the Euclidean Distance.

Thus, in the previous program which calculated the sides and angles of a right triangle, the line of code which read

```
sideC = sqrt(pow(sideA, 2.) + pow(sideB, 2.));
```

could be replaced with

```
sideC = hypot(sideA, sideB);
```

thus eliminating some function call overhead.

Generating Random Numbers

The **rand** and **srand** routines (see *rand*(3C)) exist for the generation of random numbers. **rand** is the random number generator itself, and **srand** enables you to specify a starting point (or *seed*) for **rand**.

The following program simply sets up an infinite loop and lets **rand** run for awhile (to terminate it, press Break or its equivalent):

```
main()
{
    unsigned value;

    srand(1);
    for(;;) {
        value = rand();
        printf("Random number is %u\n", value);
        sleep(1);
    }
}
```

Note that **rand** and **srand** deal only with *unsigned* integers. If you let this program run for awhile, you'll notice that the random values returned are quite large, and don't often venture below 1000. If your application requires smaller random numbers, take the value returned by **rand** modulo the range desired.

srand initializes the random number generator to a particular starting point. In the above program, 1 is used, but you can specify any positive integer you like.

The **sleep** library routine causes the program to suspend operation for the number of seconds specified (1, in this case).

Floating-Point Exponentiation Routines

Two routines, **frexp** and **ldexp** (see *frexp*(3C)), are covered in this section. **frexp** accepts a **double** value, and returns two values, **x** and **n**, such that

`value = x * 2^n`

where x is a **double** quantity of magnitude less than 1, and n is an integer exponent. **frexp**'s syntax

`frexp(value, eptr);`

where *value* is the value to be processed, and *eptr* is a pointer to an integer variable where the exponent n is to be stored. The quantity x is returned as **frexp**'s return value.

The following program accepts a number argument and uses **frexp** to output that number's representation in the form shown above:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, x, frexp();
   int eptr;

   sscanf(argv[1], "%lf", &value);
   x = frexp(value, &eptr);
   printf("%g = %g * 2^%d\n", value, x, eptr);
}
```

ldexp accepts a *value* of type **double** and an integer exponent *exp*, and returns a **double** quantity equal to

value×2^*exponent*

The following program accepts two number arguments, **value** and **exp**, and outputs the result:

```
main(argc, argv)
int argc;
char *argv[];
{
   double value, result, ldexp();
   int exp;

   sscanf(argv[1], "%lf", &value);
   sscanf(argv[2], "%d", &exp);
   result = ldexp(value, exp);
   printf("%g * 2^%d = %g\n", value, exp, result);
}
```

12

Advanced HP-UX Programming

This chapter describes how to write programs that interface with the HP-UX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution. Specifically, this chapter describes

- getting command line arguments and environment variable values from a C program
- handling errors using `stderr` and the `exit` system call
- performing input/output using low-level system calls such as `read`, `write`, and `lseek`
- managing processes using system calls such as `system` and `fork`
- handling interrupts using system calls such as `signal`

The routines described in this chapter are covered only at a general level. For details on the routines discussed here, refer to the appropriate pages in the *HP-UX Reference*.

All the examples are written in C, but you are not restricted to using C. For details on calling the routines from other languages, see the *HP-UX Portability Guide*.

Program Arguments and Environment Pointer

When a C program runs, the `main` function is passed three arguments: the number of arguments on the command line when the program was invoked, an array of pointers to the command line arguments, and a list of pointers to environment definitions strings. Traditionally, programmers have named these parameters `argc`, `argv`, and `envp`, respectively.

12

int argc

The `argc` parameter contains the number of command line arguments specified when the program was invoked. The name of the command is also counted as a comment line argument. For example, the following command line sets `argc` to 4:

```
$ cmd one two three
```

Note that invoking a program via the `exec` system call can cause the program name to *not* be passed as `argv[0]`! Programs that use `argv[0]` usually assume that it contains the program name, so this alternate invocation could cause strange failures.

char *argv[]

The `argv` parameter is an array of pointers to null-terminated strings containing command line arguments. `argv[0]` is always the name of the command as it was invoked on the command line. So, for the command line

```
$ /users/michael/bin/foo -lpp 60 < infile
```

`argc` is 3, and the elements of `argv` are set as follows:

```
argv[0] is /users/michael/bin/foo
argv[1] is -lpp
argv[2] is 60
```

Notice that the redirection symbol `<` and `infile` are *not* command line arguments. Only the arguments preceding any redirection or pipe symbol are passed to the command as arguments.

char **envp

The envp parameter is a list of pointers to environment definition strings for the process. These strings are of the form

VARIABLE=value

and are actually environment variable definitions. You can step through the strings by incrementing the envp pointer until *envp is NULL.

Example

The following C program displays argc, argv, and envp values:

```
#include <stdio.h>
main(argc, argv, envp)
     int argc;
     char *argv[];
     char **envp;
{
  int n;

  printf("Number of arguments: %d\n", argc); /* display argc */

  printf("\nArguments:\n"); /* display individual arguments */
  for (n = 0; n < argc; n++)
    printf("arg[%d] = %s\n", n, argv[n]);

  printf("\nEnvironment Strings:\n");
  while (*envp != NULL)  /* display environment strings */
    printf("%s\n", *envp++);
}
```

Compiling and running this program produced this output:

```
$ cc -o args args.c
$ args foo bar
Number of arguments: 3

Arguments:
arg[0] = args
arg[1] = foo
arg[2] = bar

Environment Strings:
_=/user/michael/bin/args
HOST=hpfcmas
HOME=/user/michael
HISTSIZE=64
SHELL=/bin/ksh
MAIL=/usr/mail/michael
  :
```

Error Handling: stderr and exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `wc` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it, so the success or failure of a program can be tested by another program that uses it as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations. The preceding example, `wc`, has only a one-exit condition, so it provides no means for detecting errors when it is used as a sub-process.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired. Use of `_exit` becomes necessary when terminating a parent and child process because both processes set up variables and buffers that are duplicates of each other. If `_exit` is not used during termination of at least one of the processes, both sets of buffers are flushed, causing duplicate output.

Low-Level Input/Output

This section describes the bottom level of input/output on the HP-UX system. The lowest level of input/output in HP-UX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

File Descriptors

In the HP-UX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a **file descriptor**. Whenever input/output is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of `READ(5, ... )` and `WRITE(6, ... )` in FORTRAN) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers are similar to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file's descriptor.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0 (`stdin`), 1 (`stdout`), and 2 (`stderr`), called the standard input, the standard output, and standard error. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal input/output without needing to open extra files.

If input/output is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

read and write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are:

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than *n* bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next new-line, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 1024, which is a convenient buffer size. Buffered 1024-byte blocks are more efficient, but one-character-at-a-time input/output is not inordinately inefficient. (Some character special files insist on reads or writes of a specified or minimum size. Refer to the appropriate *HP-UX Reference* entry for more information.)

By combining these concepts, we can write a simple program to copy from a specified input file to a specified output file. This program can copy anything to anything by specifying redirected input and output files.


```
#define BUFSIZE 1024
main()                          /* copy input to output */
{
    char buf[BUFSIZE];
    int  n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define CMASK 0377    /* for making char's > 0 */
getchar()         /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

`c` *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for Series 300 computers, but not necessarily for other computers and systems.)

The second version of `getchar` does input in big chunks, and hands out the characters, one at a time:

```
#define CMASK 0377          /* for making char's > 0 */
#define BUFSIZE  1024
getchar()                   /* buffered version */
{
    static char         buf[BUFSIZE];
    static char         *bufp = buf;
    static int          n = 0;

    if (n == 0) {           /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```


open, creat, close, unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat`.

`open` is similar `fopen`, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;
   :
fd = open(name, oflags);
```

As with `fopen`, the *name* argument is a character string corresponding to the external file name. The `oflags` argument is different. It consists of one or more flags that are logically ORed to indicate what types of file operations are to be allowed while the file is open. One of the three flags `O_RDONLY` (open for read only), `O_WRONLY` (open for write only), or `O_RDWR` (open for read/write) *must* be included. Refer to *open*(2) in the *HP-UX Reference* for a complete list of flags, some of which can be changed while the file is open. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

If you need to open a file that does not exist, use a third argument to specify the filemode as follows:

```
fd = open(name, oflags, mode);
```

12 As before, `open` returns a file descriptor if it was able to create the file called `name`, or `-1` if not. If the file already exists, `open` truncates it to zero length. `mode` defines the access mode that is to be assigned to the file if the file does not already exist.

In the HP-UX file system, `mode` defines nine bits of protection information associated with a file that control read, write, and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is convenient for specifying the permissions. For example, 0755 specifies read, write, and execute permission for the owner; and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the HP-UX utility cp, a program which copies one file to another:


```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644      /* RW for owner, R for group, others */
main(argc, argv)           /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2)           /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As mentioned earlier, there is a limit (typically 60) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine close breaks the connection between a file descriptor and an open

file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(char *`*filename*`)` removes a file from the file system. *filename* points to a null-terminated string containing the name of the file to unlink.

Random Access: lseek

File input/output is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is *fd* to move to position *offset*, which is taken relative to the location specified by *origin*. Subsequent reading or writing will begin at that position. *offset* is a `long`; *fd* and *origin* are `int`s. *origin* can be 0, 1, or 2 to specify that *offset* is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n)      /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0); /* get to pos */
    return(read(fd, buf, n));
}
```


Error Processing and errno

The routines discussed in this section and, in fact, all routines that are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1.

Routines can also specify additional information on what caused an error by setting an error code in the external variable `errno`. Routines set this variable *only when they incur errors*. Thus, a successful call to a routine does not reset the value of `errno` to zero.

The *errno*(2) page in the *HP-UX Reference* provides a detailed listing of the possible values for `errno`. In addition, individual man-pages for various library routines usually document the values that can be set in `errno` for the particular routine.

The header file `<errno.h>` (`/usr/include/errno.h`) contains symbol constant definitions for error codes returned in `errno`. Use these constants to compare against the value of `errno`, rather than comparing against hard-coded numbers. This will ensure your code is portable for subsequent releases if the values of the constants were to change for some reason.

Error constants can be used by a program, for example, to determine whether an attempt to open a file failed because it did not exist or because the user lacked permission to read it. In many cases, you may want to print the reason for failure. The routine `perror` prints a message associated with the value of `errno`. More generally, the `strerror` routine can be used to return an

error string that your program can print. For details on these routines, see *perror*(3C).

12

Processes

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

The system Function

The easiest way to execute a program from another is to use the standard library routine **system**. **system** takes one argument, a command string exactly as typed at the terminal (except for the new-line at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of **sprintf** may be useful.

Remember that **getc** and **putc** normally buffer their input; terminal input/output will not be properly synchronized unless this buffering is defeated. For output, use **fflush**; for input, see **setbuf** in the appendix.

Low-level Process Creation: execl and execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's **system** routine is based on.

The most basic operation is to execute another program *without* **returning**, by using the routine **execl**. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command including the directory path because you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program if `exec` succeeds.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is:

`execv(`*filename, argp*`);`

where *argp* is an array of pointers to the arguments; the last pointer in the array must be NULL so `execv` can tell where the list ends. As with `execl`, *filename* is the file in which the program is found, and *argp*[0] is the name of the program. (This arrangement is identical to the *argv* array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories – you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string *commandline* that contains the complete command as it would have been typed at the terminal, then call:

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, */bin/sh*. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in *commandline*.

12 In addition to the `execl` and `execv` system calls, HP-UX provides several other similar routines, including ones that search the process's environment space. These are listed on the *Programming on HP-UX Quick Reference* card. For details on these other routines, see *exec*(2).

Control of Processes: fork and wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

It splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the **process ID**. In the **child** process (that is, the newly created process), `proc_id` is zero. In the **parent process** (that is, the original process), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is:

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);      /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the *command* and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

Often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
    execl(. . .);
wait(&status);
```


This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's `system` routine.

The `status` returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

In addition to the `wait` system call, you can use the `waitpid` system call, which waits for a specific process to terminate before continuing. For details, see *wait*(2).

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up for `stdin`, `stdout`, and `stderr`, respectively. All other possible file descriptors are available for use. When this program calls another one, proper etiquette suggests making sure the same conditions hold. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

Pipes

A **pipe** is an input/output channel intended for use between two processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

12

```
$ ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int  fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error . . . */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If `O_NDELAY` is not set (see *read*(2) and *write*(2)) and a process reads a pipe which is empty, the process will wait until data arrives. If a process writes into a pipe that is too full, the process will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file. If `O_NDELAY` is set, `read` and `write` both return immediately with the value 0.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(`*cmd*`,` *mode*`)`, which creates a process *cmd* (just as `system` does), and returns a file descriptor that will either read or write that process, according to *mode*. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor *fout* will send their data to that process through the pipe.

popen first creates the the pipe with a **pipe** system call; it then **fork**s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via **execl**) to run the desired process. The parent, likewise, closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.


```
#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ?  (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int  mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);
    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);

        _exit(1); /* disaster has occurred if we get here */
    }
    if (popen_pid == -1)
        return(NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```


are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, that is, the standard input. dup is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function pclose to close the pipe created by popen. The main reason for using a separate function rather than close is that it is desirable to wait for the termination of the child process. First, the return value from pclose indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the wait lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd) /* close pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
 int  status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}
```


The calls to **signal** make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable *popen_pid*; it really should be an array indexed by file descriptor. A **popen** function, with slightly different arguments and return value is available as part of the standard input/output library discussed below. As currently written, it shares the same limitation.

Signals (Interrupts)

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals:

12

Interrupt	Sent when the Interrupt character is typed (user configurable, usually DEL)
Quit	Generated by the Quit character (user configurable, usually `File Separator` character obtained by `CTRL-\`)
Hangup	Caused by hanging up the phone
Terminate	Generated by the **`kill`** command.

Unless other arrangements have been made (see *setprgp*(2) and *signal*(2)), when one of these events occurs, the signal is sent to all processes that were started from the corresponding terminal, terminating the process(es). In the **`quit`** case, a core image file is written for debugging purposes.

The routine that alters the default action is called **`signal`**. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address, and is either a function, or a somewhat strange code that requests that the signal either be ignored or that it be given the default action. The include file **`signal.h`** gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
. . .
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination.

In all cases, **signal** returns the previous value of the signal. The second argument to **signal** may instead be the name of a **void** function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:


```
#include <signal.h>
main()
{
    void onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process . . .*/
    exit(0);
}

void onintr( )
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to **signal**? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started in the background with **&**), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the **onintr** routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that **signal** returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:


```
#include <signal.h>
#include <setjmp.h>
jmp_buf  sjbuf;

main( )
{
   int (*istat)( ), onintr( );

   istat = signal(SIGINT, SIG_IGN);    /* save original status */
   setjmp(sjbuf);                /* save current stack position */
   if (istat != SIG_IGN)
      signal(SIGINT, onintr);

   /* main processing loop */
}

onintr( )
{
   printf("\nInterrupt\n");
   longjmp(sjbuf);          /* return to saved state */
}
```

The include file **<setjmp.h>** declares the type **jmp_buf**, an object in which the state can be saved. **setsjbuf** is such an object; it is an array of some sort. The **setjmp** routine then saves the state of things. When an interrupt occurs, a call is forced to the **onintr** routine, which can print a message, set flags, or whatever. **longjmp** takes as argument an object stored into by **setjmp**, and restores control to the location after the call to **setjmp**, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling **exit** or **longjmp**, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted", the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs that catch and resume execution after signals should be prepared for errors are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, **wait**, and **pause**.) A program whose **onintr** program just sets **intflag**, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar( ) == EOF)
   if (intflag)
                /* EOF caused by interrupt */
   else
                /* true end-of-file */
```

Another aspect of error handling that must be dealt with is associated with programs where the user has elected to catch an asynchronous signal such, as an interrupt or quit signal, and the signal occurs during a system call producing the error EINTR. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned the EINTR error unless the system call is restarted. For more information, refer to *sigvector*(2).

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:


```
if (fork( ) == 0)
     execl(. . . );
signal(SIGINT, SIG_IGN);    /* ignore interrupts */
wait(&status);              /* until the child is done */
signal(SIGINT, onintr);     /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard input/output library function **system**:

```
#include <signal.h>

system(s)    /* run command string s */
char *s;
{
     int status, pid, w;
     register int (*istat)( ), (*qstat)( );

  if ((pid = fork( )) == 0) {
          execl("/bin/sh", "sh", "-c", s, 0);
       _exit(127);
     }
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     while ((w = wait(&status)) != pid && w != -1)
          ;
     if (w == -1)
          status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     return(status);
}
```


As an aside on declarations, the function **signal** obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values **SIG_IGN** and **SIG_DFL** have the right type, but are chosen so they coincide with no possible actual functions.

13

make: A Command for Maintaining Computer Programs

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. `make` provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell `make` the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the `make` command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of `make` is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the dependencies.

`make` also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

This chapter describes

- an overview of using `make`
- basic features of `make`
- the format of `Makefile`s and how to do macro substitutions
- syntax and usage of `make`
- implicit `make` rules
- an example of using `make`
- suggestions and warnings about using `make`
- suffixes and transformation rules
- using `make` with SCCS

Overview

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (such as `yacc` or `lex`). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, re-compiling all files just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependencies and command sequences is stored in a file, the simple command `make` is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the `make` command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think → edit → `make` → test ...

`make` runs on the HP-UX operating system, and is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple-source versions or of describing huge programs.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. **make** does a depth-first search of the graph of dependencies. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named **prog** is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line:


```
#include "defs"
```

The following text describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lS  -o  prog

x.o  y.o :   defs
```

Note **make** prefers tab characters instead of space characters in front of the cc or other dependent command.

If this information were stored in a file named **makefile**, the command:

```
make
```

would perform the operations needed to recreate **prog** after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

make operates using three sources of information:

- a user-supplied description file (as above)
- file names and *last-modified* times from the file system
- built-in rules to bridge some of the gaps

In our example, the first line says that **prog** depends on three *.o* files. Once these object files are current, the second line describes how to load them to

create **prog**. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, **make** discovers that there are three *.c* files corresponding to the needed *.o* files, and uses built-in information on how to generate an object from a source file (that is, issue a `cc -c` command).

The following long-winded description file is equivalent to the one above, but takes no advantage of **make**'s innate knowledge:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lS  -o  prog

x.o :  x.c  defs
        cc  -c  x.c
y.o :  y.c  defs
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files had changed since the last time **prog** was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then **prog** would be created from the new *.o* files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload **prog**.

If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry **save** might be included to copy a certain set of files, or an entry **cleanup** might be used to throw away unneeded intermediate files. In other cases one may

maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:


```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. `$` is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: `$*`, `$@`, `$?`, and `$<`. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
. . .
```

The command

```
make
```

loads the three object files with the `lS` library. The command:

```
make  "LIBES= -ll -lS"
```

loads them with both the `lex` (`-ll`) and the Standard (`-lS`) libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments containing embedded blanks in HP-UX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. Also, as in C and shell programming, any characters following a a hash mark (#) are treated as a comment and ignored, as is the hash mark itself. Blank lines and lines beginning with a hash mark are also totally ignored. If a non-comment line is too long to fit on a single source line, it can be continued to one or more subsequent lines by using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A line containing an equal sign (=) is a macro definition line. A macro definition has this syntax:

macro_name = *macro_value*

macro_name is a string of letters and digits which is replaced by *macro_value* when expanded. The *macro_name* must start in the first column; it cannot be preceded by blanks or tabs. However, the equal sign can be surrounded by any number of tabs or blanks, which are not part of *macro_name* or *macro_value*. The following are all valid macro definition lines:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the `make` command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [*target2* ...]: [:][*dependent1* ...] [; *commands*][# ...]
[Tab *commands*][''#'' ...] ...

Items inside brackets can be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters * and ? are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type:

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). **make** normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the **i** flags has been specified on the **make** command line, if the fake target name **.IGNORE** appears in the description file, or if the command string in the description file begins with a hyphen. Some HP-UX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (such as **cd** and Shell control commands) that have meaning only within a given shell process. Results from a previous line are forgotten before the next line is executed.

Before issuing any command, certain macros are set. **-$@** is set to the name of the file to be "made". **-$?** is set to the string of names that were found to be younger than the target.

If the command was generated by an implicit rule (see below), `-$<` is the name of the related file that caused the action, and `-$*` is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name `.DEFAULT` are used. If there is no such name, `make` prints a message and stops.

13

Command Usage

The `make` command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

`make` [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

`-i`	Ignore error codes returned by invoked commands. This mode is entered if the fake target name `.IGNORE` appears in the description file.
`-s`	Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name `.SILENT` appears in the description file.
`-r`	Do not use the built-in rules.
`-n`	No execute mode. Print commands, but do not execute them. Even lines beginning with an `@` sign are printed.
`-t`	Touch the target files (causing them to be up to date) rather than issue the usual commands.

`-q` Question. The `.IT` command to `make` returns a zero or non-zero status code depending on whether the target file is or is not up to date.

`-p` Print out the complete set of macro definitions and target descriptions

`d` Debug mode. Print out detailed information on files and times examined.

`-f` Description file name. The next argument is assumed to be the name of a description file. A file name of `-` denotes the standard input. If there are no `-f` arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

Implicit Rules

The **make** program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (Descriptions of these tables and means of overriding them are included at the end of this chapter.) The default suffix list is:

.o	Object file.
.c	C source file.
.e	Efl source file.
.r	Ratfor source file.
.f	Fortran source file.
.s	Assembler source file.
.y	Yacc-C source grammar.
.yr	Yacc-Ratfor source grammar.
.ye	Yacc-Efl source grammar
.l	Lex source grammar

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

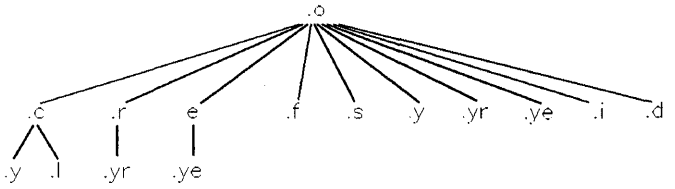


Figure 13-1. Default make Transformation Paths

If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through **lex** before compiling the result. However, if there were no *x.c* but there were an *x.l*, **make** would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

causes the newcc command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of make, we will present the description file used to maintain the make command itself. The code for make is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the make command

P = und -3  opr -r2      # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c
gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make:    $(OBJECTS)
         cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
         size make

$(OBJECTS):  defs
gram.o: lex.c
cleanup:
         -rm *.o gram.c
```

```
        -du
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make
print:  $(FILES)     # print recently changed files
        pr $?  $P
        touch print
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp  grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
lint :  dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```


make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  -c version.c
cc  -c main.c
cc  -c doname.c
cc  -c misc.c
cc  -c files.c
cc  -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the **size make** command; the printing of the command line itself was suppressed by an **@** sign. The **@** sign on the **size** command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The **print** entry prints only the files that have been changed since the last **make print** command. A zero-length file *print* is maintained to keep track of the time of the printing; the **$?** macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro:


```
make print "P = opr -sp"
```

or

```
make print "P=  cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has an **#include** *defs* line, then the object file *x.o* depends on **defs**; the source file *x.c* does not. (If **defs** is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what **make** would do, the **-n** option is very useful. The command

```
make -n
```

orders **make** to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the **-t** (touch) option can save a lot of time: instead of issuing a large number of superfluous re-compilations, **make** updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (`-d`) causes **make** to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

13

Suffixes and Transformation Rules

The **make** program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the `-r` flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name `.SUFFIXES`. **make** looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, **make** acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a *.r* file to a *.o* file is thus `.r.o`. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule `.r.o` is used. If a command is generated by using one of these suffixing rules, the macro `$*` is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro `$<` is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for `.SUFFIXES` in his own description file; the dependents will be added to the usual list. A `.SUFFIXES` line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
      (CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
      (EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
      (AS) -o $@ $<
.y.o :
      (YACC) $(YFLAGS) $<
      (CC) $(CFLAGS) -c y.tab.c
      rm y.tab.c
      mv y.tab.o $@
.y.c :
      (YACC) $(YFLAGS) $<
mv y.tab.c $@
```

Using make with SCCS

You can use make to ensure that changes to header files maintained by SCCS will cause a recompile. The following example illustrates such use:

```
SRCS=  test1.c test2.c test3.c test4.c head.h head2.h head3.h
GRPA=  test1.o test2.o test3.o test4.o
GRPB=  test1.o test3.o
GRPC=  test4.o
test:  test1.o test2.o test3.o test4.o
       cc -o test test1.o test2.o test3.o test4.o
$(GRPA): head.h
$(GRPB): head2.h
$(GRPC): head3.h
sources: $(SRCS)
$(SRCS):
         get s.$@
```

14

SCCS: Source Code Control System

This chapter describes SCCS (*S*ource *C*ode *C*ontrol *S*ystem), which is simply a set of HP-UX commands that enable you to

- track all changes made to a text file
- retrieve the current (latest) version of a file
- retrieve any previous version of a file, ignoring any changes made to the original after a given revision
- control who changes a file
- keep track of the date and location of each change made to a file along with the name of the person making the change
- add comments when indicating the reason for each change

Note SCCS does not support Access Control Lists (ACLs) and cannot be used in an environment where ACLs are used to enhance system security.

Overview

One application of SCCS is to keep track of source files during the development and maintenance of large systems. This chapter is directed towards this use of SCCS. However, it can be used in any project that involves supporting groups of related text files. Object code cannot be maintained under SCCS.

Once you store a program source file under SCCS, all of its versions, plus additional log information, are kept in a file called the **s-file**. S-files are also referred to as **SCCS files** and must have an `s.` prefix on their name. Three major operations can be performed on the s-file:

1. Get a file for some non-editing purpose, such as compilation. This operation retrieves a read-only version of the file from the s-file. By default, the latest version of the file is retrieved. This file is specifically NOT intended to be edited or changed in any way, so any changes made to a file retrieved in this way will probably be lost.
2. Get a file for editing. This operation also retrieves a version of the file from the s-file, but this file is intended to be edited and then incorporated back into the s-file. Only one person at a time can edit a particular version of an s-file at a time (unless you have specifically allowed concurrent edits on the same version).
3. Merge a file back into the s-file. This is the companion operation to (2). A new version number is assigned, and comments are saved explaining why this change was made.

Terms

You need to know the meaning of several terms before using SCCS:

S-files

An s-file is a single file that holds all the different versions of your source file. The s-file is stored in a differential format meaning that only the differences (deltas) between versions are stored, rather than the entire text of the new version. This saves disk space and makes it easy to remove selective changes later if needed. The s-file also contains header information for each version. The header also contains the comments provided by the person who created the version, explaining why the changes were made. A description of what this header information includes is presented later in this chapter.

Deltas

Each set of changes to the s-file (approximately equivalent to a version of the file) is called a delta. Although technically a delta includes only the current changes made, in practice it is usual for each delta to be made with respect to all the deltas that preceded it. This matches normal usage, where the previous changes are not saved at all and all changes are automatically based on all other changes that have happened through history. However, it is possible to get a version of the file that has selected deltas removed out of the middle of the list of changes. All of the deltas of a file maintained under SCCS are stored in an s-file.

SIDs (Version Numbers)

A SID (*S*CCS *ID*) is a number that represents a particular delta. This is normally a two-part number consisting of a release number and a level number. The form of two-part SIDs is:

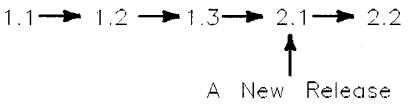
release.level

where *release* and *level* are non-zero, positive integers. Normally the release number stays the same while the *level* increments with each delta. However, you can move into a new release of a file if some major change is being made. Since all past deltas are normally applied when a given version is retrieved, the

SID of the final delta applied is used to represent the version number of the file as a whole.

Deltas applied to one SCCS file can be treated as nodes of a tree, where the initial version of the file is the root node. The root delta (node) normally has the SID number `1.1` and the deltas that follow are `1.2`, `1.3`, etc. The naming of successor deltas by incrementing the SID level number is performed automatically by SCCS when you retrieve a file for editing with `get -e`, although the delta itself is not created until you execute `delta`.

Figure 14-1 illustrates the development of an SCCS file where each delta depends on all of the previous deltas.



14

Figure 14-1. Development of SCCS File

ID Keywords

When you retrieve a version of a file from SCCS with intent to compile it (or, rather, do anything other than edit it), some special keywords are expanded by SCCS when they are found in the file. These ID keywords can be used to include the current version number or other information into the file. All ID keywords are of the form **%*x*%**, where *x* is an uppercase letter. For example, **%I%** is the SID of the latest delta applied in retrieving a particular version, **%W%** includes the module name, SID, and a string of characters that makes it accessible by the `what` command, and %G% is the date of the latest delta applied. A list of all of the ID keywords can be found in the Quick Reference section at the end of this chapter and in the entry for *get*(1) in the *HP-UX Reference*.

For example, assume that you have a source file stored under SCCS and it contains the line of code:

```
static char SccsId[] = "%W%";
```

When you retrieve the file for editing, the text file will contain the line just as it appears above. However, when you retrieve the file for compilation the **%W%** is expanded to indicate the module name, SID, and the string of characters recognized by `what`:

```
static char SccsId[] = "@(#)prog.c    1.2    05/15/84";
```

The **what** command is a valuable tool for quickly finding out information about a particular version of a program. To use it the program's source code must be contained in SCCS files. In the SCCS files, any string of information that you want to be accessed by **what** must begin with the ID keyword **%Z%**. (**%W%**, mentioned earlier, is actually a combination of several ID keywords, including **%Z%**.) When the files are retrieved for compilation, this ID keyword is expanded to the string: **@(#)**. When you invoke **what** on a file, the command prints out anything it finds between this string and the first ", >, \, *newline*, or *null* character. Refer to the section "Using ID Keywords" for more information about **what**.

When you retrieve a file for editing, the ID keywords are not expanded; this is so that after you store the file back into SCCS, they can still be expanded automatically when the file is retrieved for compilation. If you edit and store a version of a file in which the ID keywords are expanded, SCCS can no longer control the updating of the ID keywords' values. For example, if you use the ID keyword for the file's version and then store the keyword's expanded value, all of the following versions will indicate that same version number—SCCS cannot increment it. Also, if you compile a version of the program without expanding a version number ID keyword that appears in it, it is impossible to tell what version it is since all that the code will contain is **%I%**.

Creating SCCS Files

To put source files into SCCS format, use the **admin** command. The following stores a file called *s.file* under SCCS:

```
admin -ifile s.file
```

The **-i** option indicates that **admin** is to create a new SCCS file (called an s-file) and initialize its contents with the contents of the file *file*. The **s.file** argument is the name of the s-file. All s-file names *must* begin with **s.***filename*. The initial version of *s.file* is a set of changes (delta 1.1) applied to a null s-file.

After creating a new s-file, **admin** returns the message:

```
No id keywords (cm7)
```

if you have not included any ID keywords in it. This is just a warning message and it is discussed further in a later section.

Since you have stored the contents of *s.file* under SCCS, you can now remove the original file:

`rm` *file*

Note that if the name of the SCCS file is the same as the original text file except for the *s.* prefix, the original file must be removed or moved to another directory. This is because when you retrieve a version of an SCCS file, the name of the resulting text file is the SCCS file name with the *s.* removed. If there is already a writeable file with this name in your current directory, SCCS does not allow you to retrieve the SCCS file version in most cases.

14

Assume that your current HP-UX directory contains several C source files that you want to maintain under SCCS. The following shell script stores each under SCCS with the required `s.` prefix added onto its name and removes the original source files.

```
#! /bin/ksh                    specifies a Korn shell script
for i in *.c
    do
        admin -i$i s.$i
        rm $i
    done
```

If you want to have ID keywords in the files, it is best to put them in before you create the s-files. If you do not, `admin` prints "`No Id Keywords (cm7)`" after each s-file is created. If you create an s-file without ID keywords then later decide to add them, simply retrieve the file for editing, add the ID keywords, store the changes, then state that ID keywords have been added when you are prompted for comments.

Removing SCCS Files

In order to protect s-files, SCCS does not supply a direct method of removing them from your system. S-files are protected from accidental deletion in two ways:

- They are created as read-only files.
- There is no SCCS command that removes them.

Because of this protection, you must make the files writeable before you can remove them. Use **chmod** to change the access permission on an s-file:

```
chmod +w s.file
```

The **+w** indicates that you are adding write access to the file *s.file*. Once you have a writeable s-file, you can remove it using the HP-UX command:

```
rm s.file
```


Getting Files for Compilation

To get a copy of the latest version of the SCCS file *s.file*, type:

```
get s.file
```

get responds, for example, with:

```
1.1
87 lines
```

indicating that version 1.1 was retrieved and that it has 87 lines. The retrieved text is placed in a file in the current directory whose name is formed by deleting the *s.* prefix. The file is read-only to remind you that you are not supposed to change it. If you do make changes, they are lost the next time someone does a **get**.

To retrieve all of the SCCS files in a directory so that they can be compiled, specify the directory name as an argument to **get**:

get *directory*

The retrieved text files are place in your current directory and any non-SCCS files (files without the *s.* prefix) in the directory are silently ignored.

Note that if the s-file (or the directory containing s-files) that you want to access is not located in your current directory you must specify its full pathname.

Changing Files (Creating Deltas)

Getting a Copy to Edit

To edit a source file, first use `get` with its `-e` (*e* for *e*dit) option to retrieve it:

14

```
get -e s.file
```

`get` responds, for example, with:

```
1.1
87 lines
New delta 1.2
```

The retrieved file *file* (without the *s.* prefix) is placed in your current directory with read and write access permissions added to it. Edit the file using a standard text editor such as `vi`.

To retrieve all of the SCCS files in a directory for editing, specify the directory name as an argument to `get -e`:

```
get -e directory
```

Merging the Changes Back Into the S-File

When the desired changes have been made to the text file, use the `delta` command to store the changes back into the SCCS file:

```
delta s.file
```

assuming that the s-file is located in your current directory. If it is located in a different directory you must also specify a pathname for the s-file. `delta`

prompts you for `Comments?` before merging the changes into the previous version. At this time you should type a one-line description of what the changes mean (more lines can be entered by ending each line except the last with a backslash \). `delta` then responds, for example, with:

```
1.2
5 inserted
3 deleted
84 unchanged
```

saying that delta 1.2 was created, and it inserted five lines, removed three lines, and left 84 lines unchanged. (Changes to a line are counted as a line deleted and a line inserted.) Finally, SCCS removes *file* from your current directory. To retrieve it again, use `get`.

Note that the comments that you are prompted for are not maintained as part of the text body of the s-file. They are kept in another section of the s-file that is used internally by SCCS.

When To Make Deltas

In general, it is unwise to make a delta before every re-compilation or test unless other people need to edit the file at the same time. Creating too many deltas can result in unclear comments such as `fixed compilation problem in previous delta` or `fixed botch in 1.3`. However, it is very important to delta everything before installing a module for general use. A good technique is to edit the files you need, make all necessary changes and tests, compiling and editing as often as necessary without making deltas. When you are satisfied that you have a working version, `delta` everything being edited, re-`get` them, and recompile everything.

Working on a project with several people presents a problem when two people need to modify a particular version of a file at the same time. SCCS prevents this by locking the version while it is being edited (unless concurrent editing of one version has been specifically allowed). This means that you should not retrieve a file for editing unless you are actually going to edit it at that time, since you will be preventing other people on the project from making necessary changes. As a general rule, all source files that you are editing should be stored with `delta` before being used in compilations. This gives other users a better chance of being able to edit files when they need to.

What's Going On: The Sact Command

To find out who is currently editing an SCCS file, use:

```
sact s.file
```

For each editing session taking place on the file, `sact` (*S*CCS *act*ivity) tells you which SID (version) is being edited, what SID will be assigned to the new delta when editing is done, who is doing the editing, and the date and time that editing began (when `get -e` was invoked). If no one is currently editing *s.file*, `sact` returns an error message telling you that a p-file does not exist for the file (the "Types of Files" section later in this chapter discusses p-files).

You can specify more than one SCCS file name as arguments to `sact`; each file is checked one at a time. You can also specify a directory, in which case `sact` checks every SCCS file in that directory and silently ignores non-SCCS files (files without the *s.* prefix).

Using ID Keywords

ID keywords inserted into your file are expanded when you use `get` to retrieve a file for compilation. They record information about the file such as the time and date it was created, the version retrieved, and the module's name. For example, a line in an SCCS file such as:

```
static char SccsId[] = "%W%\t%G%";
```

is replaced with something like:

```
static char SccsId[] = "@(#)prog.c      1.2     08/29/80";
```

in the retrieved source file. This tells you the name and version of the source file and the time the delta was created. The string `@(#)` is the expanded form of the keyword `%Z%` and is searched for by the `what` command. (Note that the `%W%` ID keyword shown above is shorthand for several other ID keywords including `%Z%`.) Thus you can use `what` to conveniently and quickly locate expanded ID keywords in text file. Note that when you retrieve a file for editing, keywords are not expanded. This keeps them in their original form when you store the file again with `delta`.

Approximately 20 ID keywords are provided for use in SCCS files. They are listed in the Quick Reference section at the end of this chapter and in the *get*(1) entry in the *HP-UX Reference*.

The what Command

When **%Z%** is used, expanded ID keywords in files can be located using **what**. To find out the current version number of a source file and what version of it is used in an object file and final program (assuming you have previously inserted the necessary ID keywords in the SCCS source file), use:

```
what file.c file.o a.out
```

what prints all strings it finds that begin with **@(#)** in the three files. It works on all file types, including binaries and libraries. Typical output from such a commend resembles the following:


```
file.c:
        file.c  1.2     08/29/88
file.o:
        file.c  1.1     02/05/88
a.out:
        file.c  1.1     02/05/88
```

From this, it is quickly evident that the source in *file.c* does not compile into the same version as the binary in *file.o* and *a.out*.

what searches the specified files for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword. It then prints what follows that string until the first double quote ("), greater than (>), backslash (\), new-line character, or (nonprinting) null character. Note that you can locate and display constant text as well as ID keywords with **what** if you precede that text with **%Z%**.

For example, assume an SCCS file *s.prog.c* contains the following line:

```
char id[] "%Z%%M%:%I%;
```

Note that the colon (:) is not part of an ID keyword. It is left unchanged when the ID keywords are expanded. Next, the command line

```
get s.prog.c
```

is executed. The retrieved file *prog.c* is then compiled to produce *prog.o* and *a.out*. The command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
    prog.c:1.2
prog.o:
    prog.c:1.2
a.out:
    prog.c:1.2
```

indicating that version 1.2 of the file `prog.c` was used in all three files.

14

Where to Put Id Keywords

ID keywords can be inserted anywhere in SCCS files, including comments. ID keywords that are compiled into the object module are especially useful, since they let you compare what version of the object is being run to the current version of the source.

When you put ID keywords into header files, it is important that you assign them to different variables. For example, you might use:

```
static char AccessSid[] = "%W% %G%";
```

in the file *access.h* and:

```
static char OpsysSid[] = "%W% %G%";
```

in the file *opsys.h*. Had you used the same variable name in both, compilation errors would result because the variable is redefined. Also note that if you place ID keywords in a header file as code that is eventually compiled then include that same header file in multiple modules that are loaded together, the same version information will appear several times in the resulting object module. To prevent the problem, insert header file ID keywords as comments.

Creating New Releases

When you are ready to create a new release of a program, you can specify the new release number using `get`'s `-r` option. For example:

```
get -e -r2 s.prog.c
```

retrieves the latest release 1 version of *s.prog.c* and causes the next delta to be in release 2 (an SID of 2.1). Future deltas are automatically in release 2.

To assign a new release number for all of the SCCS files in a directory, use:

`get -e -r2` *directory*

assuming that the previous release was release 1, and then execute:

`delta` *directory*

All SCCS files in the directory are assigned a new delta SID of 2.1.

Canceling an Editing Session

If you retrieve a file for editing with `get -e` then decide that you do not want to edit it, cancel the editing session with:

`unget s.`*file*

`unget` returns the SID of the canceled delta. Only the person who began an editing session can cancel it. `unget` can accept more than one filename argument or, alternatively, use:

```
unget -
```

in which case `unget` accepts file names from standard input.

If you are currently editing a number of SCCS files in one directory and want to cancel all of the editing sessions for them, you can specify the directory:

`unget` *directory*

In this case `unget` checks every SCCS file in the directory. If one of the files is not currently being edited, `unget` returns an error message indicating that its

associated p-file does not exist (see "Files Used by SCCS" section later in this chapter).

If you are currently editing more than one version of a file, `unget`'s `-r` option allows you to specify which version's editing session you want to cancel:

```
unget -r2.3 s.file
```

If you find that you retrieved a file for editing when you actually needed it for some other purpose, you would like to cancel the editing session but keep the file in the current directory. Normally when you cancel an editing session, `unget` removes the retrieved text file from the current directory. You can request that it not be removed with the `-n` option:

```
unget -n s.file
```

This leaves the text file *file* still available for inspection or compilation, but any changes made to the file cannot be stored back in the SCCS file with `delta`.

You can request that `unget` execute silently (not print out the file's canceled delta's SID) using the command's `-s` option:

```
unget -s s.file
```

Restoring Old Versions

This section discusses how `get`'s `-r`, `-x`, and `-i` options are used to retrieve various versions of a file. They can be used in any combination. The `-e` option can also be used with them to create a new delta based on particular versions.

Reverting to Old Versions

Normally, `get` retrieves the latest version of the specified file. However, you can request a particular version using `get`'s `-r` option.

Suppose that after delta 1.2 was stable you made and released a delta 1.3. However, this introduced a bug, so you made a delta 1.4 to correct it. Then you found that 1.4 was still buggy, and you decided you wanted to go back to the old version. You can access delta 1.2 by choosing the SID in a `get`:

```
get -r1.2 s.prog.c
```

This produces a version of *prog.c* that is delta 1.2. Any changes that you made between delta 1.2 and the most recent delta are ignored.

If you specify a release number but not a level number, the highest level number that exists within that release is retrieved. `get -r` also allows you to retrieve particular branch deltas. Branches are discussed in the section "Maintaining Different Branches" later in this chapter.

If you try to retrieve for compilation a particular version that does not exist, SCCS responds with an error message. There is one exception: if you specify only a release number and that release doesn't exist, SCCS retrieves the delta with the highest release number that does exist, and with the highest level number within that release.

In some cases you don't know what the SID of the delta you want is. However, `get` allows you to revert to the version of the program that was running as of a certain date using its `-c` (cutoff) option. For example,


```
get -c840722120000 prog.c
```

retrieves whatever version was current as of July 22, 1984 at 12:00 noon. Trailing components can be stripped off (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line is equivalently stated with:

```
get -c"84/07/22 12:00:00" prog.c
```

Selectively Excluding Old Deltas

Suppose that you later decided that you liked the changes in delta 1.4, but that delta 1.3 should be removed. You could do this with the `-x` option:

```
get -e -x1.3 s.prog.c
```

When delta 1.5 is made, it includes the changes made in delta 1.4, but excludes the changes made in delta 1.3. You can exclude a range of deltas using a dash. For example, if you don't want to include 1.3 and 1.4 you can use:

```
get -e  -x1.3-1.4 s.prog.c
```

which excludes all deltas from 1.3 to 1.4. Alternatively,

```
get -e  -x1.3-1 prog.c
```

excludes a range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using the `-x` option (or `-i`, see below) there are conflicts between versions. For instance, it may be necessary to both include and delete a particular line, in which case SCCS always prints out a message telling the range of lines affected; these lines should then be examined very carefully to see if the version SCCS got is correct.

Since each delta (in the sense of "a set of changes") can be excluded at will, it is usually useful to put each semantically or conceptually distinct change into its own delta.

Selectively Including Deltas

Just as `get`'s `-x` option allows you to exclude deltas from a version in which they are normally included, the `-i` allows you to include deltas that are not normally included.

For example, assume that you have an SCCS file containing five deltas, 1.1 through 1.5. To retrieve a version of a file containing only deltas 1.1, 1.3, and 1.5, request that version 1.1 be retrieved and force the inclusion of deltas 1.3 and 1.5:

```
get -r1.1 -i1.3,1.5 s.file
```

To retrieve version 1.5 all of the deltas must be used. All of the following `get` command lines accomplish this.

```
get -r1.5 -i1.2 s.file
get -r1.5 s.file
get s.file
```

Note that the `-i` option in the first command line has no effect since delta 1.2 is already used to construct version 1.5. The `-r` option is not required either since delta 1.5 is the most recent delta and, by default, `get` retrieves the version incorporating it.

If there are conflicts between versions when you use the `-i` option, SCCS provides a message indicating the range of lines affected, just as it does when the `-x` option is used. You should examine these lines in the retrieved file to make sure that they are correct.

Removing Deltas

`get -x` allows you to exclude deltas from the retrieved file; however, the deltas are not removed from the SCCS file and the information they contain is still available and consuming space. To permanently remove a delta from an SCCS file, use **`rmdel`**. **`rmdel`** requires that you use the **`-r`** option to specify which delta is removed:

```
rmdel -r1.3 s.file
```

Before you can use **`rmdel`** to remove a delta, all of the following requirements must be met:

- The specified version of the file is not currently being edited.
- The SID must be the most recent delta on its branch of the delta chain for the named file: No other deltas can depend on it.
- You originally created the delta or you are the owner of the SCCS file and the directory that it is in.

The Help Command

Error messages returned by the SCCS commands have the form:

`ERROR :` *message* `(`*code*`)`

If it is not clear from *message* why the error occurred, use the associated *code* as an argument to the **`help`** command. Invoking:

`help` *code*

often provides a little more explanation about the cause of the error. For example, if you execute **`get`** *program* you could receive the following message:

`ERROR[`*program*`]: not an SCCS file (co1)`

Executing:

```
help co1
```

produces:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Auditing Changes

The prs Command

When you create deltas, you presumably give reasons for the deltas in response to the `comments?"` prompt. To print out these comments later, use:

14

```
prs s.file
```

Note that `prs` provides information about each of the deltas used to create the requested version of the file; therefore, it is a way to list the deltas upon which a particular version depends. It produces a report for each delta providing the time and date of creation, the user who created the delta, and the comments associated with the delta. For example, the output of the above command might be:

```
s.file:

D 1.3 84/04/12 08:21:35 becky 3 2  00020/00008/00021
MRs:
COMMENTS:
inserted 20 lines, removed 8 lines

D 1.2 84/04/11 09:21:08 becky 2 1  00008/00000/00021
MRs:
COMMENTS:
inserted 8 lines

D 1.1 84/04/10 06:37:14 becky 1 0  00021/00000/00000
MRs:
COMMENTS:
```

```
date and time created 84/04/10 06:37:14 by becky
```

The report indicates that the file's initial delta (created with `admin -i`) inserted 21 lines, delta 1.2 inserted 8 lines and left 21 unchanged, and delta 1.3 inserted 20 lines, removed 8 lines, and left 21 lines unchanged.

You can request information about a particular version of a file using `prs`'s `-r` option:

```
prs -r2.3 s.prog.c
```

`prs` can accept multiple file names or directory names as arguments. If you request information about all of the SCCS files in a directory, you should probably redirect `prs`'s output to a file and look at it at your leisure:

`prs` *directory* `>`*output*

When a directory is specified, the effect is as if each SCCS file it contains were named and any non-SCCS files are ignored.

`prs` also allows you to modify the information it provides using its `-d` option. Refer to the `prs` entry in the *HP-UX Reference* to see how this is done.

Determining Why Lines Were Inserted

To find out why you inserted various lines in a file, you can get a copy of the file with each line preceded by the SID of the delta that created it using:

```
get -m s.prog.c
```

where the retrieved copy is called *prog.c*. Once you have determined which delta inserted the line you are interested in, use `prs` to find out what that particular delta did by looking at its comment line.

Another way to find out which lines were inserted by a particular delta (e.g., 1.3) is:

```
get -m -p s.prog.c | grep '^1.3'
```

The `-p` flag causes `get` to output the retrieved text to the standard output rather than to a file.

Comparing Versions

To compare two versions of a file, use **sccsdiff**. For example,

```
sccsdiff -r1.3 -r1.6  s.prog.c
```

outputs the differences between delta 1.3 and delta 1.6 in a format similar to the format used by the **diff** command.

You can specify any number of file names with **sccsdiff** but the same two SIDs specify which versions are compared for all of them. You cannot specify a directory as an argument.

Files Used by SCCS

14

As a user of SCCS, you do not need to know all of the information covered in this section; however, it should give you a feel for the inner workings of SCCS.

There are 8 types of files that are used by SCCS and all of them are ASCII text files. They are:

S-files	SCCS files created by **admin -i**.
G-files	Text files containing the "body" of SCCS files and created by **get**.
L-files	Files containing delta dependency information and created by **get -l**.
P-files	Files created and used by SCCS to keep track of multiple edits.
D-files	Temporary files created and used by SCCS during the execution of **delta**.
Q-files	Temporary files created and used by SCCS to update p-files.
X-files	Temporary files created and used by SCCS to update s-files.
Z-files	Lock-files created and used by SCCS to prohibit simultaneous updating of s-files.

Normally, only 4 of these file types are visible to users of SCCS: s-files, g-files, l-files, and p-files. The remaining 4 types are temporary files used internally by SCCS during the execution of particular commands.

S-Files

S-files are often referred to as SCCS files in this chapter. They contain all of the versions of files you are maintaining under SCCS. You create and name an s-file when you initially enter a file into SCCS:

```
admin -ifile s.file
```

`s.`*file* is the new s-file and *file* can now be removed. Accessing a file maintained under SCCS using SCCS commands is done using its s-file name. S-file names must begin with the prefix `s.`.

The Contents of the S-File

S-files are composed of lines of ASCII text arranged in the following 6 parts:

Checksum	A line containing the *logical* sum of all the characters of the file, not including the checksum itself.
Delta Table	Information about each delta, such as type, SID, data and time of creation, and user inserted comments.
User Names	A list of login names and/or group IDs of users who are allowed to modify the file by adding or deleting deltas. Use `admin` to modify.
Flags	Indicators that control certain actions of various SCCS commands. Use `admin` to modify.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file. Use `admin` to modify.
Body	The actual text that is being administered by SCCS, mixed with internal SCCS control lines. Use `get -e` and `delta` to modify.

The Body section of the s-file is modified whenever you create or delete deltas. Use the `admin` command to modify User Names, Flags, and Descriptive

Text sections (see the "System Protection Using `admin`" section later in this chapter). The Checksum and Delta Table are modified internally by SCCS.

Since the entire contents of an s-file is ASCII, the file can be processed with various HP-UX commands, such as `vi`, `grep`, and `cat`. This is convenient but somewhat risky in those instances where an SCCS file must be modified manually (such as when the time and date of a delta are recorded incorrectly because the system clock was set incorrectly) or when you simply want to look at its contents.

Note If you modify an SCCS file directly (instead of using SCCS commands), the Checksum value may be incorrect, causing an error whenever you try to retrieve a version of the file. This problem is discussed in a later section, "Restoring the S-File". Do not edit an s-file directly unless you thoroughly understand its format.

G-Files

The `get` command creates a text file that contains a particular version of an s-file, obtained by applying deltas to the initial version. This text file is called a `g-file` and its name is formed by removing the SCCS file's `s.` prefix. It is this file that you use for inspection, compilation, or editing purposes.

G-files are created in the current directory and are owned by the real user. Their file mode depends on how `get` is invoked. If you use:

```
get s.file
```

the resulting g-file *file* has mode 444 (read only) and is produced for inspection or compilation, but not for editing. Note that any ID keywords in the file are expanded to their appropriate values.

If you use:

```
get -e s.file
```

then *file* can be edited. Note that any ID keywords in the file are not expanded, allowing them to be stored back in the file when you use `delta`.

L-Files

When retrieving an SCCS file with `get`, you can request that an `l-file` be created. Use the command's `-l` option:

```
get -l s.file
```

The name of an l-file is formed by replacing the `s.` prefix of the SCCS file with `l.`. It contains a table indicating what deltas were used to create the retrieved version of an SCCS file. You must specifically request the creation of l-files with `-l`. `get` does not create them by default.

To send delta dependency information to standard output instead of placing it in an l-file, use:

```
get -r2.3 -lp s.file
```


P-Files

When you retrieve an SCCS file for editing (`get -e`), besides creating a writeable g-file containing the version's text, a `p-file` is also created. The name of a p-file is formed by replacing the `s.` prefix of an SCCS file with `p.`.

P-files are used internally by SCCS to keep track of multiple edits on the same SCCS file (see "Concurrent Editing"). For each edit that is in progress on a particular SCCS file (`get -e` has been executed but not the associated `delta`), the file's p-file keeps track of

- the SID of the retrieved version
- the SID that will be given to the new delta when `delta` is executed
- the login name of the user that executed `get -e`
- the date and time that the `get -e` was executed

If a p-file is accidentally destroyed, it can be regenerated with:

```
get -e -g s.file
```

The `-e -g` combination suppresses the retrieval of a writeable text file (g-file), but the associated p-file is created. A p-file must exist for an SCCS file before you can use `delta` on it.

When you use the `sact` to request information, the data is obtained from a p-file.

D-Files

`D-files` are used internally by SCCS during the execution of `delta` to hold a temporary copy of the original retrieved g-file before any editing was done. The name of a d-file is formed by replacing the `s.` prefix of the associated SCCS file with `d.`. When you retrieve an SCCS file for editing (`get -e`) and then invoke `delta`, SCCS creates a d-file and compares the edited g-file with the contents of the d-file to determine what has changed. These changes are then stored in the SCCS file (s-file).

When you invoke `delta`, you can request that the differences between the d-file and the g-file (the file that you retrieved and the file that you are now storing) be sent to standard output using:

14

```
delta -p s.file
```

Once `delta` is executed, you can request the same information with the `sccsdiff` command.

Q-Files

A `q-file` is a temporary copy of a p-file that is used internally by SCCS. Its name is formed by replacing the `p.` prefix of the p-file with `q.`. Whenever a p-file needs to be updated (because editing of a version of a file was completed with `delta` or started with `get -e`), a q-file is first created. The change is made to the q-file and then the p-file is removed and the q-file is renamed to become the new p-file. This strategy is used to ensure the integrity of the p-file in case there are any problems adding or deleting entries from the table.

X-Files

An `x-file` is a temporary copy of an s-file that is used internally by SCCS. All SCCS commands that modify an SCCS file do so by first creating and modifying an x-file. This ensures that the SCCS file is not damaged if the processing terminates abnormally. The name of this temporary copy is formed by replacing the `s.` prefix of the SCCS file with `x.`. When processing is complete, the old s-file is removed and the x-file is renamed to be the s-file.

Z-Files

`Z-files` are lock-files SCCS uses to prevent simultaneous updating of an SCCS file. They are discussed later in this chapter in the section "SCCS Protect Facilities".

Concurrent Editing

Concurrent Edits on Different Versions

SCCS allows different versions of one SCCS file to be edited at the same time. This means that a number of `get -e` commands can be executed on the same file provided that no two executions retrieve the same version, unless concurrent edits on the same version are allowed (see the discussion in the next section).

SCCS uses a p-file to keep track of the edits that are in progress on one file. The first execution of `get -e` causes the creation of a p-file for the specified SCCS file. Subsequent executions of the command update the p-file, adding entries in the file for each edit session that is in progress. Each entry in the p-file specifies the SID of the retrieved version, the SID that will be assigned to the new delta, and the login name of the person doing the editing. When an editing session is terminated (with `delta` or `unget`), the corresponding entry in the file's p-file is removed. If no other versions of the file are currently being edited, then the p-file itself is removed.

Before SCCS allows an editing session on a particular version of an SCCS file to begin, it makes sure that if a p-file for the file already exists there is no entry in it specifying that the version has already been retrieved. If there is no entry with that SID, SCCS adds an entry for the new editing session. If there is an entry with the same SID, SCCS generates an error message and does not allow the version to be retrieved for editing (unless multiple edits of the same version are allowed). SCCS informs you if editing is currently being done on another version of the file you request to edit.

Note

Multiple executions of `get -e` must be done from different directories. This is because each time any version of one file is retrieved, the resulting g-file (text file) is assigned the same name. As a result, SCCS prohibits multiple edits on the same file in the same directory because the g-file would constantly be overwritten.

In practice, multiple editing sessions are performed by different users with different working directories; therefore, this restriction normally does not cause a problem.

Concurrent Edits on the Same Version

By default, SCCS does not permit multiple executions of `get -e` on the same version of one SCCS file. Each editing session on a version begun with `-e` must be ended with `delta` before another session can begin. However, you can allow concurrent edits *by a single user* on the same version of a file by setting the file's `j` flag with the `admin` command (see "System Protection Using admin" later in this chapter).

Note that if you do set a file's `j` flag, multiple editing sessions on the same version must be done in different directories, just like multiple edits on different versions. In addition, these edits can only be performed by the first user to check out the file. Traditionally, users needing to edit the same `s.file` concurrently (like several working on a joint project) have done so by using a single login account (perhaps a project name) to do this.

Recovering from Problems

Making Temporary Changes

If you use `get -e` to retrieve a file so that you can edit it, SCCS requires that you delta the changes that you make back into the associated s-file. Sometimes, however, it is necessary to make modifications to a file that you do not want saved.

To make temporary changes to a file possible, retrieve it from SCCS with:

```
get s.file
```

SCCS does not expect changes to be made to the file; therefore, it gives it read-only access. You must now change the mode of the file so that you can edit it:

```
chmod +w file
```

`Chmod +w` adds write access to a file. Any changes that you now make to *file* cannot be stored in SCCS.

Recovering an Edit File

Sometimes you may find that you have lost a file that you were trying to edit. Unfortunately, you can't just execute `get -e` again; SCCS keeps track of the fact that someone is trying to edit that version, so it won't let you do it again. Neither can you retrieve it using `get`, since that would expand the ID keywords. Instead, you can say:

```
get -k prog.c
```

This retrieves the file and does not expand the ID keywords, so it is safe to do a delta with it.

Restoring the S-File

You may find that the SCCS file itself is corrupt. The most common way this happens is when someone edits the file directly, not through the SCCS commands. SCCS keeps a checksum that contains the *logical* sum of all of the characters in the file. If you modify the SCCS file directly the checksum may have the wrong value. No SCCS command will process a corrupted SCCS file except `admin -h` and `admin -z` as described below.

You should audit all SCCS files for corruption on a regular basis. The simplest way to do this is to execute `admin` using the `-h` option on all of the SCCS files of interest:

```
admin -h s.file1 s.file2 ...
```

or:

```
admin -h directory
```

This checks to see if each file's checksum is correct. The message `corrupted file (c06)` is produced for a file whose checksum is not correct.

If you have a corrupted SCCS file, you must first determine why its checksum is incorrect. If it is due to someone having directly modifying the file, the problem is often corrected by merely recomputing the checksum. Do this with `admin`'s `-z` option:

```
admin -z prog.c
```

The checksum is recomputed to bring it into agreement with the actual contents of the file.

Note

Before using `admin -z`, first find and correct the corruption problem. If you don't, once the checksum is recomputed, the corruption is no longer detectable. `Admin -z` does not find or fix the problem, it merely recomputes a new checksum.

Using the Admin Command

The `admin` command creates new SCCS files and changes parameters of existing ones. When an SCCS file is created, its parameters are either initialized with options or are assigned default values if no options are specified.

Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file can use `admin` on it.

Creating SCCS Files

As discussed earlier, an SCCS file for a file called *prog* is created using:

```
admin -iprog s.prog
```

The name of the SCCS file is *s.prog*. If no file name is specified with the `-i` option, the text is read from standard input:

```
admin -i s.prog prog
```

When the SCCS file is created, the release number assigned to its initial delta is normally `1` and the level number is always `1`, meaning that the first delta of the file is `1.1`. You can assign a different initial release number using `admin`'s `-r` option when the file is created:

```
admin -iprog -r3 s.prog
```

Here, the initial delta is `3.1`.

Adding Comments to Initial Delta

When you create an SCCS file, you can supply a comment stating the reason for the creation of the file. This is done with the `-y` option:

```
admin -ifile -y"The reason this file was created" s.file
```

If you do not specify an initial comment with `-y`, SCCS gives the initial delta a comment line of the form:

`date and time created` *YY/MM/DD HH:MM:SS* `by` *logname*

Descriptive Text in Files

A portion of an SCCS file is reserved for descriptive text, text that summarizes the content and purpose of the SCCS file. When you are creating an SCCS file you can insert descriptive text using `admin`'s `-t` option followed by the name of a file containing the text:

```
admin -ifile -tdescrip s.file
```

You can either add descriptive text to an existing SCCS file or replace the descriptive text it already contains with:

```
admin -tnew_descrip s.file
```

where *new_descrip* is the name of the file containing the descriptive text. To remove descriptive text from an SCCS file, use `-t` without a file name:

14

```
admin -t s.file
```

To see the descriptive text for an SCCS file, use `prs` as follows:

```
prs -d:FD: s.file
```

The `prs` command's `-d` option allows you to specify what information about the file that you want returned. The `:FD:` indicates that you want to see the file's descriptive text. Refer to the *HP-UX Reference* manual entry for `prs` for more information about the command's `-d` option.

Setting SCCS File Flags

SCCS files have a number of parameters called `flags` that can be added and deleted using the `admin` command. These flags are maintained in a particular section of SCCS files along with their associated values where appropriate. Add flags with `admin`'s `-f` option and delete them with its `-d` option. For example:

```
admin -fd2.1 prog.c
```

sets the `d` flag to the value `2.1`. This flag can then be deleted using:

```
admin -dd prog.c
```

You can use `admin -f` to add or `admin -d` to delete the following flags:

`b` Allow branches to be made using `get -e -b`.

d*SID*

Default *SID* to be used on a `get`. If this is just a release number, the default is the highest version number for that release.

c*ceiling*

Sets the highest release number for a file that can be retrieved with `get -e` to *ceiling*. *ceiling* must be a number less than or equal to 9999. The default release ceiling for a file is 9999.

f*floor*

Sets the lowest release number for a file that can be retrieved with `get -e` to *floor*. *floor* must be a number greater than 0 and less than 9999. The default release floor for a file is 1.

i

Give a fatal error during `get` or `delta` if there are no ID keywords in a file. This is useful to guarantee that a version of the file does not get merged into the s-file that has the ID keywords inserted as constants instead of internal forms.

j

Allow concurrent edits on the same version (SID) of the SCCS file.

l*list*

A *list* of releases that cannot be retrieved for editing (`get -e`). The *list* has the following syntax:

list = *range* | *list*, *range*

range = *RELEASE_NUMBER* | `a`

The character `a` is equivalent to specifying all of the releases for the named SCCS file. If you do not specify a *list* with the `l` flag, `a` is assumed by default.

To delete one or more "locked" releases with `admin`'s `-d` option you must also use a *list* to specify which releases are to be "unlocked". For example, `admin -dla s.file` unlocks all of the releases of *s.file* so that they can be edited.

n

Causes `delta` to create a term|null| delta in each of those releases (if any) being skipped when a delta is made in a new release (for example, when making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created

from them. If this flag is not set for a file, skipped releases are non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* — Replace all occurrences of the ID keyword **%Q%** with the contents of file *text* when the SCCS file is retrieved for inspection or compilation. If the **q** flag has not been set for a file, occurrences of **%Q%** are not replaced with anything.

m*module* — Replace all occurrences of the ID keyword **%M%** with the specified *module* name when the SCCS file is retrieved for inspection or compilation. If the **m** flag has not been set for a file, occurrences of **%M%** are replaced with the name of the SCCS file minus the **s.** prefix.

t*type* — Replace all occurrences of the ID keyword **%Y%** with the specified *type* when the SCCS file is retrieved for inspection or compilation. If the **t** flag has not been set for a file, occurrences of **%Y%** are not replaced with anything.

v[*pgm*] — Causes **delta** to prompt for Modification Request (MR) numbers as the reason for creating a delta. If you set this flag when you create an SCCS file, **admin**'s **-m** option must also be specified, even if its value is null.

You can optionally specify an MR number validation checking program called *pgm* with **admin -fvpgm**.

Specifying Who Can Edit a File

admin's **-a** option allows you to specify who can edit an SCCS file. Use it as follows:

```
admin -alogin s.file
```

where *login* is a user's login name or an HP-UX group ID. If it is a group ID, the effect is equivalent to specifying all login names common to that group ID. Several **-a** options can be used on a single **admin** command line.

Note that **admin** can accept one or more SCCS file names or directory names as arguments. For example, the command line:

```
admin -abill -ajane -ajohn directory
```

gives HP-UX users **bill**, **jane**, and **john** editing privileges to all of the SCCS files in *directory*. The list of users for each SCCS file in the directory is updated to show this. No one else can edit those SCCS files unless specifically authorized by using **admin -a**.

If no one has been assigned editing privileges to a file with **admin -a**, the file's list of users is empty and anyone can edit the file (as long as they have write access to the file's parent directory).

To remove a user's ability to edit an SCCS file, use **admin -e**. For example:

```
admin -ebill directory
```

removes bill from the list of users allowed to edit the SCCS files in *directory*.

Note

Before a user can be prohibited from editing a file, the file's list of users must be non-empty. If the list is empty everyone has editing privileges and using `admin -e` has no effect.

If a file's list of users is non-empty, any user not added to the list with `admin -a` is already prohibited from editing the file. Thus, you can remove a specific user's editing privileges only if you have previously added him to the list of users using `admin -a`.

Maintaining Different Branches

14 Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production. This can be done using a "branch." Normally deltas continue in a straight line, each depending on the delta before. Creating a branch "forks off" a version of the program.

For example, Figure 14-2 shows there is one branch delta having an SID of 2.1.1.1:

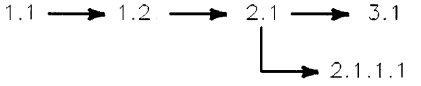


Figure 14-2. Example Branch Delta

The ability to create branches off of the latest main "trunk" delta must be enabled in advance by setting the file's `b` flag:

```
admin -fb prog.c
```

The `b` flag can also be set when the SCCS file is first created. It is not necessary to set a file's `b` flag in order to create a branch off of an older delta.

Creating a Branch

To create a branch off of the latest main trunk version, use:

```
get -e -b prog.c
```

If the retrieved version has an SID of 1.5 and no branch was previously created on it, a branch with SID 1.5.1.1 is created when the file is modified. The deltas for this branch are numbered 1.5.1.n where "n" increments by 1 with each delta.

If you retrieve an old version of an SCCS file for editing, SCCS automatically assigns a branch SID to the new delta. The file's `b` flag need not be set to do this. For example, assuming that the latest delta of prog.c is delta 1.5 you can create a branch off of delta 1.2 using:

```
get -e -r1.2 prog.c
```


SCCS will automatically number the new branch delta 1.2.1.1 if it is the first branch off delta 1.2.

Retrieving a Branch

Deltas in a branch are not normally included when you use `get`. To retrieve these versions, you have to use:

```
get -r1.5.1 prog.c
```

specifying the requested branch's SID.

Branch Numbering

SCCS uses the following SID numbering scheme for recognizing branch deltas:

release.level.branch.sequence

release.level is the SID of the delta on the main trunk from which the branch descends. A *branch* number is assigned to each branch path that originates from a particular delta on the main trunk. A *sequence* number is assigned to each delta on a particular branch. Branch deltas always have all four of the above components in their SIDs and the release and level numbers are always those of the ancestral main trunk delta.

When you retrieve a branch, specifying only the release, level, and branch components of the SID returns the most recent version on a particular branch.

Although SCCS maintains enough internal information to remember delta dependencies of branch deltas, the SID number itself does not always indicate all of the deltas between a branch delta and its main trunk ancestor delta. For example, given delta 1.3.2.2 you know that the main path ancestor is delta 1.3 and that it is the second delta (sequence=2) on the second branch (branch=2) descending from delta 1.3. However, the diagrams below indicate two possible development paths for delta 1.3.2.2:

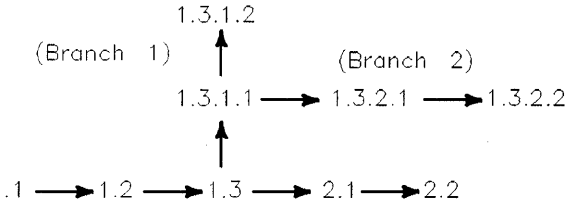


Figure 14-3. Diagram 1

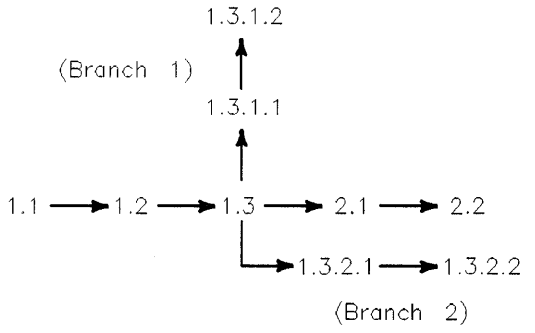


Figure 14-4. Diagram 2

Note that in Diagram 1, version 1.3.2.2 is dependent on deltas 1.1, 1.2, 1.3, 1.3.1.1, and 1.3.2.1, while in Diagram 2 the delta with the same SID is dependent on 1.1, 1.2, 1.3, and 1.3.2.1.

A Warning

Branches should be kept to a minimum. After the first branch from the main trunk, SIDs are assigned rather haphazardly, and the structure gets complex very quickly.

SCCS Protection Facilities

The protection facilities that SCCS provides for a system fall into two categories:

- general protection of files inherent to SCCS (using general HP-UX file system protection by appropriately setting the modes of various files)
- specific system protection strategies controlled by the SCCS administrator by using the `admin` command

General File Protection

New SCCS files created with `admin` are given mode 444 (read only). This mode prevents any direct modification of the files by any non-SCCS commands. The mode of the files should not be changed to allow direct modification.

SCCS files must not be linked to more than one filename because of the way SCCS modifies files. Commands that modify SCCS files (`delta`, `admin`) create a copy of the file. The copy, called an x-file, is modified; the original SCCS file is removed, and the copy is renamed. If the original SCCS file has any links, they are broken when it is removed. SCCS generates an error message if you try to process any file under SCCS that has multiple links.

To prevent simultaneous updates to SCCS files, a `lock-file` (called the **z-file**) is also created whenever an x-file is created. A z-file contains the process number of the command that created it, and its existence is an indication to other commands that the SCCS file is being updated. Other SCCS commands that modify SCCS files will not process an SCCS file if a corresponding z-file exists. For example, assume that two people are editing two versions of an SCCS file. When one of them executes `delta`, a z-file is created which keeps the second person from successfully invoking `delta`. When `delta` has

completed, the z-file is removed and the second person is free to create his own delta. z-files are created with mode 444 (read only) in the directory containing the SCCS files and are owned by the effective user.

SCCS checks for the corruption of an SCCS file by maintaining a checksum. Whenever the file is modified with an SCCS command, its checksum is updated to reflect the logical sum of the number of characters the file has. Most SCCS commands will not allow you to access a file that is corrupted. The `admin` command allows you check for corrupted file and to correct them.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files since most of the commands allow you to operate on all of the SCCS files in a directory by merely specifying a directory name. The contents of directories should correspond to convenient logical groupings, such as subsystems of a large project.

14

System Protection Using admin

`admin` allows the system administrator of a project to control five major areas of protection:

1. prohibiting concurrent editing of a given version of a file
2. specifying a list of users that have permission to edit a file
3. prohibiting editing on particular releases
4. setting range limits to what releases users can access
5. making the recognition of no ID keywords in a file by SCCS commands a fatal error

The `admin` command allows you to use these protection strategies on either a file-by-file basis or on a directory basis. For details on how to do this, see "Using the Admin Command" earlier in this chapter.

Using SCCS With Make

If you are using **make** to create and maintain systems and are using SCCS to maintain the source files for the systems, you can make the two work together by including SCCS commands in **make**'s makefiles. The following discussion assumes that you already know how to use **make**. For more information, refer to the *make*(1) entry in the *HP-UX Reference* or Chapter 13 in this book.

Most makefiles should have a few basic target entries:

`a.out`	(or whatever the makefile generates). This target entry regenerates whatever this makefile is supposed to regenerate. If the makefile regenerates several intermediate things, this should be called "all" and should in turn have dependencies on everything the makefile can generate.
install	Moves the objects to their final resting place, doing any special **chmod**'s or **ranlib**'s as appropriate.
sources	Creates all the source files from SCCS files.
clean	Removes unneeded files from the directory.

The clean entry should not remove files that can be regenerated from the SCCS files since it is sufficiently important to have the source files around at all times.

Note that the examples of makefiles that follow are only partial and do not illustrate all of these target entries fully. Also note that the example makefiles require that you execute **make** in the same directory as the SCCS files.

To Maintain Groups of Programs

Frequently there are directories with several largely unrelated programs (such as simple commands) and these can often be maintained by one makefile. For example, the makefile below maintains "prog" and "example":

```
LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.c prog.h
```

```
example: example.o
        $(CC) $(LDFLAGS) -o example example.o
example.o: example.c

.DEFAULT:
        get s.$<
```

Note that the source for the programs is maintained as SCCS files and that these files must exist in the same directory as the makefile for the makefile to be able to retrieve them. The `.DEFAULT` rule is called every time something is needed that does not exist, and no other rule exists to make it. The explicit dependency of the *.o* file on the *.c* file is important. Another way of doing the same thing is:


```
SRCS=   prog.c prog.h example.c

LDFLAGS= -i -s

prog: prog.o
        $(CC) $(LDFLAGS) -o prog prog.o
prog.o: prog.h

example: example.o
        $(CC) $(LDFLAGS) -o example example.o

sources: $(SRCS)
$(SRCS):
        get s.$@
```

There are some advantages to the second approach:

- The explicit dependencies of .o files on .c files are not needed.
- There is an entry called *sources* so if you just want to get all the sources you can just say `make` *sources*.
- The makefile is less likely to do confusing things since it won't try to get things that do not exist.

To Maintain a Library

Libraries that are largely static are best updated using explicit commands, since **make** doesn't know about updating them properly. However, **make** can adequately handle libraries that are in the process of being developed. One problem in maintaining libraries is that the object (*.o*) files must be kept out of the library as well as in the library.

```
# configuration information
JOBJS=   a.o b.o c.o d.o
SRCS=    a.c b.c c.c d.s x.h y.h z.h
TARG=    /usr/lib

# programs
GET=     get
REL=
AR=      -ar
RANLIB= ranlib

lib.a: $(OBJS)
        $(AR) rvu lib.a $(OBJS)
        $(RANLIB) lib.a

install: lib.a
        cp lib.a $(TARG)/lib.a
        $(RANLIB) $(TARG)/lib.a

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) s.$@

print: sources
        pr *.h *.[cs]
clean:
        rm -f *.o
        rm -f core a.out $(LIB)
```

The `$(REL)` in the `$(SRCS)` entry allows you to retrieve various versions of the SCCS files. For example:

```
make  REL=-r1.3
```

Note that for the install entry to execute properly, no one should be editing any of the SCCS files when it is invoked.

To Maintain a Large Program

Consider this example makefile:

```
OBJS=   a.o b.o c.o d.o
SRCS=   a.c b.c c.y d.s x.h y.h z.h

GET=    get
REL=

a.out: $(OBJS)
        $(CC) $(LDFLAGS) $(OBJS) $(LIBS)

sources: $(SRCS)
$(SRCS):
        $(GET) $(REL) s.$@
```


(The `print` and `clean` entries are identical to the previous case.) This makefile requires copies of the source and object files to be kept during development. It is probably also wise to include lines of the form:

```
a.o: x.h y.h
b.o: z.h
c.o: x.h y.h z.h
z.h: x.h
```

so that modules are recompiled if header files change.

Since `make` does not do transitive closure on dependencies, you may find in some makefiles lines like:

```
z.h: x.h
        touch z.h
```

This would be used in cases where file z.h has a line:

```
#include "x.h"
```

in order to bring the date of *z.h*'s last modification in line with the date of the last modification of *x.h* (or rather, when the system thinks *z.h* was last modified). Alternatively, the effect of the `touch` command can be achieved by doing a `get` on *z.h*.

Using SCCS on a Multi-User Project

This section describes the how SCCS is configured to maintain files for a large project that involves several users. The person that configures and controls the SCCS files is called the "SCCS System Administrator". Only you need the information covered in this section if you are your project's SCCS System Administrator.

If you plan to use SCCS on a project that involves several users, first develop a system of controlling access to the SCCS files and commands. Thus far, this chapter has only discussed a one-user system, where that user has write access to the directory containing the SCCS files. The user also has full use of all of the SCCS commands and can modify protected files (by first making read-only files writeable).

As an SCCS System Administrator, you should provide an interface program that gives temporary write access to the SCCS directory when users execute certain SCCS commands, but restricts users to read-only access at all other times. When SCCS files are used on a project, they are grouped in one directory (or more if necessary). The SCCS System Administrator is the owner of the SCCS directory, has write access to it, and has full use of all of the SCCS commands. Other users involved on the project should only have read access to the directory, which means that they cannot directly use the SCCS commands that require write access.

The SCCS interface program is a C program that provides a filter for the commands requiring that the user have directory write access. If, instead of using the interface program, you give all of the users write access to the SCCS directory, you greatly restrict the protection facilities SCCS provides. Use of the interface provides users with only temporary write access when they execute one of the commands. The two SCCS commands that require directory write access and that must be available to the users through the interface program are `get` and `delta`. `rmdel`, `cdc`, and `unget` also require write access

and can also be made available to users through the program. The remaining SCCS commands either do not require write access to the SCCS directory or are usually used only by the SCCS System Administrator (`admin` for example).

How the SCCS Interface Works

The SCCS interface program invokes a specified SCCS command and causes the command's process to inherit the privileges of the SCCS System Administrator for the duration of its execution. This allows the process to obtain write access to the SCCS directory.

The names of the commands that you want filtered through the interface program must be linked to the program so that invoking the command name executes the program. The interface program is written in C and when a C program is executed, the name that invoked the program is passed as argument 0 and is followed by any user-supplied arguments. By looking at the value of argument 0, the program knows which command to execute. Thus, the command name used to invoke the interface program determines which SCCS command the program executes. How other arguments, such as SCCS file names, are processed is often system dependent, but they can be passed directly to the SCCS command by the program.

Configuring an SCCS System Using the Interface

As the SCCS System Administrator, there are six basic steps to carry out before allowing other users to access SCCS files:

1. Create and move to an SCCS directory.
2. Write and compile the interface program.
3. Change the mode of the program.
4. Set up links between the program and the SCCS command names.
5. Modify each user's search path so that the directory containing the interface program is searched before */usr/bin*, the directory containing the SCCS commands.
6. Create the SCCS files.

Creating the SCCS Directory

Before you can successfully use the SCCS interface program, you must create one or more directories for storing the SCCS files and the program. You, as the SCCS System Administrator, should be the only one with write access to the directory.

For example, to create a directory called */system/sccs* and then deny write access to all but yourself, use:

```
mkdir  /system/sccs
chmod  755  /system/sccs
```

Now move to the SCCS directory since you must be in that directory when writing and maintaining the SCCS interface program:

```
cd  /system/sccs
```

Writing and Compiling the Program

The SCCS interface program is written in C and this section assumes that you already know how to program in that language.

Write an SCCS interface program that is customized to the needs of your system. To get started, here is a general-purpose interface program:

```
#define LENGTH 100          /* length of command string */
main(argc, argv)
int argc;
char *argv[];
{
 register int i;       /*counts command line arguments*/
 character cmdstr[LENGTH];     /*holds SCCS command name*/
 /*
  * Do any required processing of file name arguments that
  * follow the SCCS command name (arguments that don't begin
  * with -)
  */
 for (i = 1; i<argc; i++)
     if (argv[i][0] != '-')
         argv[i] = filearg(argv[i]);
 /*
  * Get "simple name" of name used to invoke this program
  * (i.e. strip off directory-prefix name, if any).
  * This step may not be needed in your system.
  */
 argv[0] = sname(argv[0]);
 /*
  * Invoke actual SCCS command, passing arguments.
  */
 sprintf(cmdstr, "/usr/bin/%s", argv[0]);
 execv( cmdstr, argv);
}
```

This example program calls two routines that you must supply and that allow you to customize the SCCS interface. `filearg` acts as a preprocessor for SCCS commands. In the program above, it is used to modify SCCS file name. This modification often involves appending the path name of an SCCS directory to

the SCCS file names so that users can access the files without having to specify full path names.

The second routine that you must supply is **sname**. Its purpose is to modify the name with which the user invoked the interface program so that it agrees with the name of the associated SCCS command. The statement calling this routine is not required when the link names of the interface program are the same as the names of the SCCS commands.

Once you have written an SCCS interface program designed for your system, compile it. Assuming that your source code file is called *interface.c*, use the following to compile it:

```
cc interface.c -o interface
```

The name of the resulting executable program is *interface*.

Specifying Program Access Permissions

The interface program must be owned by the SCCS System Administrator, and must be executable by the other users involved on the project. It must also have its **set user ID on execution** bit enabled so that when the program is executed, the user obtains write access to the SCCS directory. Assign these necessary characteristics to the program with:

```
chmod 4755 interface
```

where "interface" is the name of the executable interface program.

Assign Name Links to the Program

Now that you have an executable interface program, use the **cp** command to assign name links to it. It is convenient for the users if these name links are the same as the SCCS commands that are executed by the program.

To illustrate, assume that you want to allow users to access the **get** and **delta** commands through the interface program. Create the necessary links with:

```
cp interface get

cp interface delta
```

You now have three names that point to the same program: *interface*, *get*, and *delta*. All of the other SCCS commands that require write access to the SCCS

directory will be inaccessible to the users since you have not linked them to the program.

Modifying the Users' Search Path

Once you have linked the appropriate SCCS command names to the SCCS interface program, you must modify each user's HP-UX search path so that the directory containing the the interface program is found before the actual SCCS commands. `PATH` is the HP-UX variable that specifies where the system looks for a command when a user executes it. When any command is executed, the system searches for the command in the directories defined by the user's `PATH` variable. The directories are searched in the order in which they appear in the variable's list. Your HP-UX system has a default definition for `PATH` but it can be redefined by each user in his *.profile* file. Refer to your system's *HP-UX System Administrator Manual* for more information about the `PATH` variable and the ".profile" file.

Whether you have to change the `PATH` variable in every user's *.profile* file or just the system's default definition, you must insert the SCCS directory name before the appearance of */usr/bin*, the directory containing the SCCS commands, in `PATH`'s directory list. For example, if a user's `PATH` variable is defined as:

```
PATH=/bin:/usr/bin
```

you should change it to:

```
PATH=/bin:/system/sccs:/usr/bin
```

where */system/sccs* is the name of the SCCS directory containing the SCCS interface program. When you execute a command, the system first searches for it in */bin*, then in */system/sccs*, and finally in */usr/bin*.

Creating SCCS Files

As SCCS System Administrator, you are the only user able to execute `admin` because it requires write access to the SCCS directory and you did not specify it as a link name to the SCCS interface program. Having sole access to `admin` means that you can strictly control the creation of SCCS files and the setting to their various flags. Refer back to the section "SCCS's Protection Facilities" in this chapter for more information.

Note that in order to make full use of SCCS for a multi-user project, SCCS files should be maintained in a central location and logically grouped into one or more SCCS directories.

Quick Reference

Commands

In the discussion of the following SCCS commands, only the most useful options are discussed. Refer to the *HP-UX Reference* for complete descriptions of the commands and all of their options.

`get`	Gets files for compilation (not for editing). ID keywords are expanded. Note that `get -e` is listed separately.
`-r`*SID*	Version to get.
`-p`	Send text to standard output rather than to the actual file.
`-k`	Don't expand ID keywords.
`-i`*list*	List of deltas to include.
`-x`*list*	List of deltas to exclude.
`-m`	Precede each line with SID of creating delta.
`-c`*date*	Don't apply any deltas created after date.
`get -e`	Gets files for editing. ID keywords are not expanded. Should be matched with a `delta` command.
`-r`*SID*	Same as `get -r`*SID*. If *SID* specifies a release that does not yet exist, the highest numbered delta is retrieved and the new delta is numbered with *SID*.
`-b`	Create a branch.
`-i`*list*	Same as `get -i`*list*.
`-x`*list*	Same as `get -x`*list*.

`delta` Merge a file retrieved with `get -e` back into the s-file. Collect comments about why this delta was made.

`unget` Remove a file previously retrieved with `get -e` without merging the changes into the s-file.

`prs` Print information about the SCCS file.

`sact` Determine who is currently editing a file.

`what` Find and print ID keywords that have been expanded. They must be preceded by `@(#)` (the expanded form of the keyword %Z%).

`admin` Create or set parameters on s-files.

`-i`*file* Create s-file, using file as the initial contents.

14 `-z` Rebuild the checksum in case the file has been corrupted.

`-f`*flag*[*value*] Turn on the flag and optionally give it a value.

`-d`*flag* Turn off (delete) the flag.

`-t`*file* Replace the descriptive text in the s-file with the contents of *file*. If *file* is omitted, the descriptive text is deleted from the s-file. Useful for storing documentation or "design and implementation" documents to insure they get distributed with the s-file.

`-h` Check for corruption in the s-file.

Useful flags are:

- `b` Allow branches to be made using the `-b` flag to get `-e`.
- `dSID` Default SID to be used on a get.
- `i` Cause `No Id Keywords` error message to be a fatal error rather than a warning.
- `t` The module *type*; the value of this flag replaces the **%Y%** keyword.

`sccsdiff` Compare two versions of an SCCS file.

`cdc` Change the comment line or MR number associated with a previously created delta.

`rmdel`	Remove a delta from an SCCS file. This delta must be the most recent on its branch or the main trunk – no other deltas can depend on it.
`help`	Supplies additional information about an SCCS error message.

ID Keywords

`%Z%`	Expands to `@(#)` for the `what` command to find. Every ID keyword string that you want `what` to see must be preceded by this keyword.
`%M%`	The current module name; for example, *prog.c*. Unless set by `admin`, it defaults to the file name minus the `s.` prefix.
`%F%`	The SCCS file name.
`%Y%`	The value of the `t` flag as set by `admin`.
`%I%`	The SID of the retrieved text. The highest delta applied.
`%W%`	A shorthand for `%Z%%M% <tab> %I%`.
`%E%`	The date of the delta corresponding to the `%I%` keyword (YY/MM/DD).
`%G%`	The date of the delta corresponding to the `%I%` keyword (MM/DD/YY).
`%U%`	The time the delta corresponding to the `%I%` keyword was created (HH:MM:SS).
`%R%`	The current release number—that is, the first component of the `%I%` keyword.
`%L%`	The current level number—that is, the second component of the `%I%` keyword.
`%B%`	The current branch number—that is, the third component of the `%I%` keyword, if it exists.
`%S%`	The current sequence number—that is, the fourth component of the `%I%` keyword, if it exists.
`%D%`	The current date (YY/MM/DD).
`%H%`	The current date (MM/DD/YY).

`%T%` The current time (HH:MM:SS).

`%Q%` The value of the `q` flag as set by **admin**.

`%C%` The current line number. It is intended for identifying messages output by the program such as **this shouldn't have happened** errors. It is *not* intended to be used on every line to provide sequence numbers.

15

The M4 Macro Processor

The `m4` macro processor is a general-purpose front end, similar in use to `Ratfor` (for FORTRAN) and `cpp` (for C). It could be used as a front-end for any programming language, including assembly language. The `#define` statement in C language and the analogous `define` in `Ratfor` are examples of the basic facility provided by `m4`. This chapter describes

- a basic overview of `m4`
- `m4` command usage
- defining `m4` macros
- arguments to macros
- arithmetic functions
- file manipulation
- executing HP-UX commands
- conditionals
- string manipulation
- printing to `stderr`

Overview of m4 Capabilities

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the `m4` macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions

The basic operation of `m4` is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be re-scanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is re-scanned.

15

The user also has the capability to define new macros. Built-in macros and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process. A list of 32 built-in macros provided by the `m4` macro processor can be found in Table 15-1.

Table 15-1. Built-in Macros

Macro Name	Function
`changequote`	Restores original characters or sets new quote characters.
`changecom`	Changes left and right comment markers from the default # and newline.
`decr`	Returns the value of its argument decremented by 1.
`define`	Defines new macros.
`defn`	Returns the quoted definition of its argument(s).
`divert`	Diverts output to 1-out-of-10 diversions.
`divnum`	Returns the number of the currently active diversion.
`dnl`	Reads and discards characters up to and including the next newline.
`dumpdef`	Dumps the current names and definitions of items named as arguments.
`errprint`	Prints its arguments on the standard error file.
`eval`	Prints arbitrary arithmetic on integers.
`ifdef`	Determines if a macro is currently defined.
`ifelse`	Performs arbitrary conditional testing.
`include`	Returns the contents of the file named in the argument. A fatal error occurs if the filename cannot be accessed.
`incr`	Returns the value of its argument incremented by 1.
`index`	Returns the position where the second argument begins in the first argument.

Table 15-1. Built-in Macros (continued)

Macro Name	Function
`len`	Returns the number of characters that makes its argument.
`m4exit`	Causes immediate exit from `m4`.
`m4wrap`	Pushes the exit code back at final EOF.
`maketemp`	Facilitates making unique file names.
`popdef`	Removes current definition of its argument(s) exposing any previous definitions.
`pushdef`	Defines new macros but saves any previous definition.
`shift`	Returns all arguments of shift except the first argument.
`sinclude`	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
`substr`	Produces substrings of strings.
`syscmd`	Executes the HP-UX System command given in the first argument.
`sysval`	Return code from the last call to *syscmd*.
`traceoff`	Turns macro trace off.
`traceon`	Turns the macro trace on.
`translit`	Performs character transliteration.
`undefine`	Removes user-defined or built-in macro definitions.
`undivert`	Discards the diverted text.

Usage

To use the `m4` macro processor, input the following command:

`m4` [*optional_files*]

Each argument file is processed in order. If there are no arguments or if an argument is "-", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following command:

`m4` [*files*]`>` *outputfile*

Defining Macros

The primary built-in function of `m4` is `define`. `define` is used to define new macros. The following input:

`define(`*name*`,`*stuff*`)`

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *stuff* is any text that contains balanced parentheses. *stuff* may stretch over multiple lines. Thus, as a typical example:

```
define(N, 100)
. . .
if (i > N)
```

defines N to be 100 and uses the symbolic constant N in a later `if` statement.

The left parenthesis must immediately follow the word `define` to signal that `define` has arguments. If a user-defined macro or function name is not followed immediately by (, it is assumed to have no arguments. Macro calls have the following general form:

`name(`*arg1*`,`*arg2*`,...`*argn*`)`

A macro name is only recognized as such if it appears surrounded by non-alphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though the variable contains a lot of Ns.

Macros may be defined in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100. If N is redefined and subsequently changes, M retains the value of 100, not N.

The **m4** macro processor expands macro names into their defining text as soon as possible. The string N is immediately replaced by 100. Then the string M is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

15

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when the value of M is requested later, the result is the value of N at that time (because the M will be replaced by N, which will be replaced by 100).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off.

Note The direction of the single quote marks is important. The left (opening) quote is different from the right (closing) quote. See **changequote** if this is a problem on your terminal.

The value of a quoted string is the string stripped of the quotes. If the input is:

```
define(N, 100)
define(M, 'N')
```

the quotes around the N are stripped off as the argument is being collected. The result of using quotes is to define M as the string N, not 100. The general rule is that **m4** always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
define' = 1
```

Another example of using quotes is redefining N. To redefine N, the evaluation must be delayed by quoting:

```
define(N, 100)
...
define('N', 200)
```

In **m4**, it is often wise to quote the first argument of a macro. The following example will not redefine N:

```
define(N, 100)
...
define(N, 200)
```

The N in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100,200)
```

This statement causes an error since only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed using the **changequote** function. The following example substitutes the opening and closing square brackets for the opening and closing single-quote characters:

```
changequote([, ])
```

The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-in macros related to **define**. The **undefine** macro removes the definition of some macro or function as follows:

```
undefine('N')
```

The macro removes the definition of N. Built-in macros can be removed with **undefine**, as follows:

```
undefine('define')
```

But once removed, the definition cannot be re-used.

The function **ifdef** provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('S310', 'define(wordsize,16)')
ifdef('S550', 'define(wordsize,32)')
```

15

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. Example:

```
ifdef('hpux', on HPUX, not on HPUX)
```

Arguments

So far, the simplest form of macro processing has been discussed, which is replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its `define`), any occurrence of `$`*n* is replaced by the *n*th argument when the macro is actually used. Thus, the macro `bump` defined as:

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The `bump(x)` statement is equivalent to `x = x + 1`.

A macro can have as many arguments as needed, but only the first nine are accessible (`$1` through `$9`). The macro name is `$s0` although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, `cat(x, y, z)` is equivalent to `xyz`. Arguments `$4` through `$9` are null since no corresponding arguments were provided. Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:


```
define(a, b c)
```

defines `a` to be `b c`.

Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is `a`. The second is literally `(b,c)`. A bare comma or parenthesis can be inserted by quoting it.

Arithmetic Functions

`m4` provides three built-in functions for doing arithmetic on integers (only). The simplest is `incr`, which increments its numeric argument by 1. The function `decr` decrements by 1. Thus, to handle the common programming situation where a variable is to be defined as "one more than N," use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then N1 is defined as one more than the current value of N.

The more general mechanism for arithmetic is a function called `eval`, which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are:

Precedent	Operator
Higher	unary + and -
	** (exponentiation)
	* / % (modulus)
	+ -
	== != < <= > >=
	! ~ (logical not and bitwise not)
	& (bitwise and)
	| ^ (bitwise or and exclusive or)
	&& (logical and)
Lower	(logical or)

Parentheses may be used to group operations where needed. All the operands of an expression given to `eval` must ultimately be numeric. The numeric value of a true relation (such as `1>0`) is 1 and false is 0. The precision in `eval` is 32 bits under the HP-UX operating system.

As a simple example, define M to be 2==N+1 using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

File Manipulation

A new file can be included in the input at any time by the built-in function **include**. For example:

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The function **sinclude** (silent include) says nothing and continues if the file named cannot be accessed.

The output of **m4** can be diverted to temporary files during processing, and the collected material can be output upon command. **m4** maintains nine of these diversions, numbers 1 through 9. If the built-in macro:

divert(*n*)

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command, which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The function **undivert** brings

back all diversions in numerical order. The function **undivert** with arguments brings back the selected diversions in the order given. If the current diversion is not 0 through 9, the act of undiverting will discard text in the specified stream(s).

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* re-scanned for macros. The function **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

System Command

Any program in the local operating system can be run by using the **syscmd** function. For example:

```
syscmd(date)
```

on the HP-UX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**. To facilitate making unique file names, the function **maketemp** is provided with specifications identical to the system function **mktemp**. The **maketemp** macro replaces the string **XXXXXX** (6 **X**'s) anywhere in the argument with the process ID of the current process.

15

Conditionals

Arbitrary conditional testing is performed via the **ifelse** function. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes, which prevents evaluation of `ifelse` from occurring too early. If the fourth argument is missing, it is treated as empty.

The function `ifelse` can actually have any number of arguments and provides a limited form of multi-way decision capability. In the input:

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

String Manipulation

The `len` function returns the length of the string (number of characters) that makes up its argument. Thus:


```
len(abcdef)
```

is 6, and `len((a,b))` is 5.

The function `substr` can be used to produce substrings of strings. Using input, `substr(`*s*`,` *i*`,` *n*`)` returns the substring of *s* that starts as the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. The call:

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time
```

If *i* or *n* is out of range, various actions occur.

The function `index(`*s1*`,` *s2*`)` returns the index (position) in *s1* where the string *s2* occurs or -1 if it does not occur. As with `substr`, the origin for strings is 0.

The function **translit** performs character transliteration and has the general form:

translit(*s*, *f*, *t*)

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So:

```
translit(s, aeiou)
```

would delete vowels from *s*.

There is also a function called **dnl** that deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up **m4** output. Using input:

15

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the function **dnl** is added to each of these lines, the newlines will disappear. Another method of achieving the same results is to input:

```
divert(-1)
define(...)
...
divert.
```

Printing

The **errprint** function writes its arguments out on the standard error file. An example would be:

```
errprint('fatal error')
```

The function **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

Glossary

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.

bss segment
A segment of memory in which uninitialized data is stored. Compare with "text segment" and "data segment." (For details, refer to *How HP-UX Works: Concepts for the System Administrator.*)

buffer
A temporary holding area for data. Buffers are used to more efficiently perform input/output.

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."

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 *How HP-UX Works: Concepts for the System Administrator*.)

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 "supporting library."

depth-first order
Searching a list starting at the end of the list and moving toward the head.

(Note that shared library initialization routines are invoked by traversing the list of loaded shared libraries depth-first.)

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).

entry point
The location at which a program starts running after HP-UX loads it into memory to begin execution.

explicit loading
The process of using the *shl_load*(3X) function to load a shared library into a running program.

export stub
Generated by the Series 700/800 linker for a global definition in a shared library.

export symbol
A symbol definition that is 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.

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 via 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
Generated by the Series 700/800 linker for external references to shared library routines.

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 via 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 lower-level than the source code, but higher level than the object code.

I-SOM
Intermediate code-System Object Module. Used during profile-based optimizations.

library
A file containing object code for subroutines and data that can be used by programs.

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.

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 *How HP-UX Works: Concepts for the System Administrator* 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*.

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. (For details, refer to *How HP-UX Works: Concepts for the System Administrator.*)

PBO
See "profile-based optimization."

PC-relative
A form of machine-code addressing in which addresses are referenced relative to the program counter 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. 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. (For details, refer to *How HP-UX Works: Concepts for the System Administrator.*)

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.

s-file
An SCCS file that has an `s.` prefix and that contains version and update log information for a file managed by SCCS.

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.

stream
A data structure of type `FILE *` used by various input/output routines.

stub
(Series 600/700/800) A short code segment that may be inserted into procedure calling sequences by the PA-RISC linker. Stubs are used for very specific purposes, such as inter-space (e.g., shared-library) calls, long branches, and preserving calling interfaces across modules (e.g., parameter relocation). Refer to the manual *PA-RISC Procedure Calling Conventions Reference Manual*.

supporting library
A library that was specified on the command line when building a shared library (with `ld -b ...`). See "dependency."

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 *How HP-UX Works: Concepts for the System Administrator*.)

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.

z-file
A lock file used by SCCS to prevent simultaneous updates..

Index

B

C

D

E

F

G

M

Q

R

S

T

U

V

W

X

Y

Z

Printed in USA E0892

Reorder No. or
Manual Part No.
B2355-90026

**Manufacturing
Part No.
B2355-90026**

B2355-90026