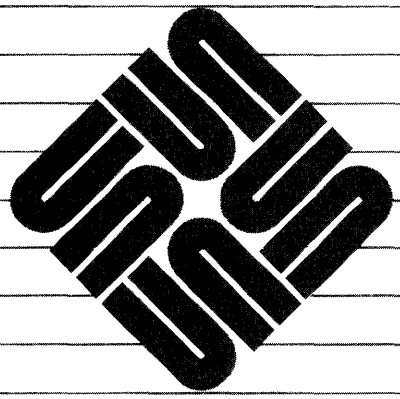
sun microsystems

Programming Utilities & Libraries



Part Number: 800-3847-10
Revision A of 27 March, 1990

Trademarks

SunOS™, Sun Workstation®, as well as the word "Sun" followed by a numerical suffix, are trademarks of Sun Microsystems, Incorporated.

UNIX® and UNIX System V® are trademarks of Bell Laboratories.

PDP-11® is a trademark of Digital Equipment Corporation.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Contents

Tables

Figures

Preface

The following chapters describe a number of system facilities, utility commands, and libraries of primary interest to application developers.

- Chapter 1: Shared Libraries

  This chapter describes Sun's approach to shared library support, along with techniques for using and creating shared libraries.

- Chapter 2: Lightweight Process Library

  this chapter describes Sun's implementation of lightweight processes.

- Chapter 3: System V Interprocess Communication Facilities

  This chapter describes facilities that support standard System V IPC.

- Chapter 4: SCCS — Source Code Control System

  SCCS is a version control utility for source files.

- Chapter 5: `make` User's Guide

  `make` is a utility that provides consistent generation of programs and systems.

- Chapter 6: `lint` — a Program Verifier for C

  `lint` is a utility that you can use to check your C programs for internal consistency and portability.

- Chapter 7: Performance Analysis

  This chapter describes system utilities for timing, profiling and coverage analysis of programs.

- Chapter 8: `m4` — a Macro Processor

  `m4` is a parametric macro-language (pre)processor.

- Chapter 9: `lex` — a Lexical Analyzer Generator

  `lex` is a program generator that produces scanning routines in C.

- Chapter 10: `yacc` — Yet another Compiler Compiler

  `yacc` is a program generator that produces parsing routines in C.

- Chapter 11: The `curses` Library

  This chapter describes the `curses` screen-cursor motion library package derived from BSD.

- Chapter 12: System V `curses` and `terminfo`

  This chapter describes the standard System V `curses` terminal-display library routines and support facilities.

- Appendix A

  This appendix summarizes the enhancements made to Sun's version of the `make` utility.

For detailed information about system utilities, library functions, file- and device-level facilities, and other details about specific features of the operating system, refer to the *SunOS Reference Manual.*

Bibliography and Acknowledgements

This manual has been derived in large part from sources that include technical papers distributed with U.C. Berkeley's BSD release, System V Release 3 documentation, and others. In particular, Sun Microsystems wishes to acknowledge the following sources:

1. Aho, A. V., and Corasick, M. J., *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM **18**, 333-340 (1975).

2. Allman, Eric, *Source Code Control System*, University of California at Berkeley.

3. Arnold, K. C. R. C., *Curses — Screen Updating and Cursor Movement Optimization: A Library Package*, Bell Laboratories, Murray Hill, New Jersey.

   Author's Acknowledgements:

   This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database [and] implement optimal cursor movement ... Doug Merritt and Kurt Shoens also were extremely important, as were ... Ken Abrams, Alan Char, Mark Horton and Joe Kalash.

   Editor's Note:

   The `curses` library was implemented by Ken Arnold, based on the screen-updating and optimizing routines originally written by Bill Joy for the `vi` editor.

4. Bonanni, L. E., and Salemi, C. A., *Source Code Control System User's Guide*, Bell Laboratories, Piscataway, New Jersey.

5. Feldman, S. I., *Make — A Program for Maintaining Computer Programs* Bell Laboratories, Murray Hill, New Jersey.

6. Graham, S. L., Kessler, P. B., and McKusick, M. K., *Gprof — A Call Graph Execution Profiler*, Computer Science Division, Electrical Engineering and

Computer Science Department, University of California at Berkeley.

Editor's Note:
This paper is for the scholar inertested in the theory behind call-graph profiling.

7. Johnson, S. C., *'A Portable Compiler: Theory and Practice', Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
8. Johnson, S. C., *Lint, a C Program Verifier* Bell Laboratories, Murray Hill, New Jersey.
9. Johnson, S. C., *Yacc — Yet Another Compiler-Compiler*, Bell Laboratories Computing Science Technical Report #32, July 1978.
10. Johnson, S. C., and Ritchie, D. M., *'UNIX Time-Sharing System: Portability of C Programs and the UNIX System', Bell System Technical. Journal* **57**(6) pp. 2021-2048 (1978).
11. Kernighan, B. W., and Plauger, P. J., *Software Tools*, Addison-Wesley, Inc., 1976.
12. Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, N. J. (1978).
13. Kernighan, B. W., and Ritchie, D. M., *The M4 Macro Processor*, Bell Laboratories, Murray Hill, New Jersey.

Author's Acknowledgements:
We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of *m4* has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code. The *m4* macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer.

14. Kernighan, B. W., *UNIX for Beginners — Second Edition*, Bell Laboratories, 1978.
15. Kernighan, B. W., and Ritchie, D. M., *UNIX Programming*, Ritchie, Bell Laboratories, Murray Hill, New Jersey.
16. Lesk, M. E., *Lex — A Lexical Analyzer Generator*, Computing Science Technical Report #39, October 1975.

Author's Acknowledgements:
[The] outside of `lex` is patterned on *yacc* and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of *lex*, as well as debuggers of it. Many thanks are due to both. The current version of `lex` was designed, written, and debugged by Eric Schmidt.

1

Shared Libraries

Operating systems like SunOS have long achieved more efficient use of memory by sharing a single physical copy of a program's `text` (code) among the processes executing it. But while the text of a program may be shared among its concurrent invocations, a significant portion of that text, consisting of library routines, may be duplicated as part of *other* running programs. For example, widely-used library functions such as `printf()` may be replicated any number times throughout memory, and again in various executables throughout the file system. This suggests that still-greater efficiencies can be had by sharing text at the *library* level whenever possible.

The SunOS shared library mechanism improves resource utilization in a way that is both straightforward and flexible:

- No specialized kernel support is required; it uses the standard memory-mapping and copy-on-write features provided by the `mmap`(2) system call and the kernel memory management facilities.
- It is designed to minimize the burdens placed on users of existing code. In particular:
  - Shared libraries are transparent to the programs that use them, as well as the build procedures for those programs.
  - They are largely transparent to standard system utilities, including debuggers.
  - Shared libraries are transparent to library source code written in C. However, some special procedures are necessary when building the shared libraries themselves.
  - The allocation of address space for shared library routines is handled automatically.
  - Unlike statically-linked executables, programs that rely on shared libraries need not be rebuilt if an underlying library changes (so long as that library's calling interface remains compatible).
  - The use of shared libraries is not required. You can specify the static version of a SunOS shared library as desired.
  - Shared libraries may be bound and unbound dynamically, during the course of program execution.

In addition, shared libraries enhance the development environment by making it easier to modify and test compatible updates to library functions.

1.1. Definitions

Shared Object

A *shared object*, or `.so` file, is an `a.out`(5) format file produced by `ld`(1). A shared object differs from a runnable program in that it lacks an initial entry point. At run-time, such an object may be linked to a number of executing programs, all of which share access to a single copy of that object.

Shared Library

A *shared library* is a shared object file that is used as a library. In cases where the shared library exports initialized data, the shared object (`.so`) may be paired with an optional *data interface description* (`.sa`) file. (See *Building a Shared Library*, below, for details.)

Static vs. Dynamic Link Editing

Link editing is the set of operations necessary to build an executable program from one or more object files. *Static linking* indicates that the results of these operations are saved to a file. *Dynamic linking* refers to these same link-edit operations when performed at run-time; the executable that results from dynamic linking appears in the running process, but is not saved to a file.

Position Independent Code (PIC)

Position-Independent code (PIC) requires link editing only to relocate references to objects that are external to the current object module. Position-independent code is readily shared.

Static and Dynamic Link Editors

The link-editing facilities of `ld` have been made available for use at run-time as well as at compile-time. At compile time, the static link editor, `ld`, can build an executable file in which some symbols remain unresolved. An executable (`a.out`) file that contains unresolved symbols is said to be *incomplete*. Incomplete executables require dynamic link editing at run-time.

The dynamic link editor, `/usr/lib/ld.so`, uses the system's memory management facilities to map in and bind the shared object files that are required at run-time, and performs the link editing operations that were deferred by `ld`. As long as the text bound-in at run-time is not subsequently modified (say, by a link-edit operation or an update to initialized external data), it remains shared among the various (disparate) programs that use it. However, if the text of a shared routine should need to be modified by a process during the course of execution, local (exclusive) copies of the affected pages are created and maintained.

1.2. Using Shared Libraries

For the application developer, the decision to use shared libraries is made at the static linking phase, when running `ld`. By default, if a shared version of a library is available, `ld` constructs an executable that uses the shared version.

Building a Program to Use Shared Libraries

`ld` combines a variety of object files to produce an executable (`a.out`) file. Exactly what code gets produced, and how complete the `a.out` is, depends on the command-line options and input files supplied as arguments on the command line. `ld` simply defers the resolution of any symbols that remain after it has run out of definitions, and assumes that the program will be fully linked by `ld.so` at run-time. `ld` accepts as input:

- Simple object files. `ld` simply concatenates (and links) `.o` files in the order that they are encountered.
- `ar`(1) libraries. Each `.a` file is searched exactly once as it is encountered, and only those definitions that match an unresolved external symbol are extracted, concatenated to the text (or data), and linked.
- Shared objects. Any `.so` encountered is searched for symbol definitions and references, but does not normally contribute to the concatenated text (see *Binding of PIC with non-PIC*, for exceptions having to do with `ld`'s `-dc` option). However, the occurrence of each shared object is noted in the resulting `a.out` file; this information is used by `ld.so` to perform dynamic link editing at run-time.

`ld`'s output can be one of two basic types:

- An "executable" (`a.out`) file. This file is either a *program*, if it has an entry point, or a *shared object* (`.so`), if it does not.
- Another "simple object" (`.o`) file. When given the `-r` flag, `ld` combines the input object files to form a single, larger one. (This is a special use for `ld` which is of little relevance to shared libraries.)

You can indicate which libraries are to be used by supplying a `-l`*name* option on the `ld` command line for each. `ld` searches each library in the order specified. The *name* string is an abbreviated version of the library's filename; the full name is of the form '`lib`*name*`.a`' if in archive format, or '`lib`*name*`.so.`*version*' if it is in shared object form. (see *Version Control* below, for a detailed discussion of the *version* suffix). At `ld`-time, this version information is noted; it must be matched properly for successful binding at run-time by `ld.so`.

The location of the library specified by a `-l` option is determined by an ordered list of directories in which to search called the library search path. This search path is specified as follows. At compile time, directories specified by the `-L` options are searched first, followed by those specified in the `LD_LIBRARY_PATH` environment variable (a colon-separated list of pathnames), and then the default libraries, `/usr/lib`, `/usr/5lib` and `/usr/local/lib`. At run-time, directories in `LD_LIBRARY_PATH` environment variable are searched first, followed by libraries specified with `-L`, and finally, the default directories.

Each directory supplied with `-L` is recorded for use when the program is executed, as are the default directories. Directory search information obtained from `LD_LIBRARY_PATH` is *not* recorded in this manner. However, the search path that `LD_LIBRARY_PATH` contains at run-time *is* searched at that time; this allows an alternate set of libraries to be used.

At `ld`-time, the library search is satisfied by the first occurrence of either form of the library (`.so` or `.a` if no `.so` is found), but if both versions are found in the *same* directory, the `.so` form is used by default. However, the choice of whether a `.so` or `.a` version is used by `ld` can be controlled by the binding mode options described in the next section.

Binding Mode Options

`-Bstatic` and `-Bdynamic`

You can specify the binding mode by supplying one of the `-B`*keyword* options on the command line:

`-Bdynamic` Allow dynamic binding, do not resolve symbolic references, and allow creation of execution-time symbol and relocation information. This is the default setting. Note that `ld` records the name of the `.so` file with the highest version number in the executable.

`-Bstatic` Force static binding, this mode is also implied by options that generate non-sharable executable formats.

`-Bdynamic` and `-Bstatic` may both be specified a number of times to toggle the binding mode for specific libraries. Like `-l`, their influence is dependent upon their location in the command line. Libraries that appear after a `-Bstatic` are linked statically. Libraries that appear after a `-Bdynamic` are treated as shared (when a shared version is available).

NOTE *Since* `-Bdynamic` *is the default setting, the use of shared libraries in the construction of a program thus "falls out" from installing the* `.so` *in* `ld`*'s library search path.*

If `-Bstatic` is in effect, `ld` refuses to use the `.so` form of a library; it continues searching for an equivalent library with the `.a` suffix, and an explicit request to load a `.so` file is treated as an error.

The following example shows how `-Bstatic` and `-Bdynamic` can be used to use selected shared and static libraries. This `cc` command:

```
cc -o test test.c -Bstatic -lsuntool -lsunwindow -Bdynamic -lsunwindow -lpixrect
```

generates the `ld` command:

```
/bin/ld -dc -dp -e start -X -o test /usr/lib/crt0.o test.o -Bstatic -lsuntool \
    -Bdynamic -lsunwindow -lpixrect -lc
```

Since `-Bstatic` turns off the use of shared libraries, `ld` finds the static (`.a`) `suntool` library and uses it for link editing immediately. The subsequent `-Bdynamic` option tells `ld` to use shared versions of the `sunwindow`, `pixrect` and C libraries, if available.

`-N` and `-n` Options for `ld`

The `ld` options `-N` and `-n` instruct `ld` to build a non-pageable executable. Their use implies a `-Bstatic` option.

Binding of PIC with Non-PIC

`-dc` and `-dp` Options

As noted in the above example, the `cc` command generates an `ld` command with the `-dp` and `-dc` options. These options are included to facilitate binding of non-PIC code (generated by default) with the PIC shared libraries that a program might use. The bindings of interest are to:

- **commons,** (`externs`): allocated after the program is completely assembled (`-dc`);
- **initialized data**: imported from the shared libraries (`-dc`); and
- **entry points**: supplied by the shared libraries (`-dp`).

Without special handling, references to these objects would require execution-time link editing, resulting in unsharable code. To improve the degree of sharing for such programs, `-dc` and `-dp` force the allocation of commons and the creation of aliases for library entry points, respectively. These allocations and aliases are created as part of the non-PIC executable, and result in programs that are considered to be "pure-text" non-PIC programs, even though they may require dynamic link editing.

NOTE *While it is possible to invoke the* `ld` *command directly, it is generally better practice to rely on the compiler-driver (such as* `cc`*) to generate the appropriate* `ld` *command, so as to remain insulated from any future changes in the compilation environment. Compiler commands such as* `cc` *accept and pass on options to* `ld`.

Use of Assertions

The `-assert` Option

To help detect any potential sharability or correctness problems, `ld` can validate certain assertions about an executable that it builds. This assertion checking is invoked by the "`-assert` *keyword*" option, where *keyword* is one of:

`definitions` if the resulting program were run now, there would be no run-time undefined symbol diagnostics. This assertion is set by default, and is sufficient for validating applications that make use of shared libraries.

`pure-text` the resulting executable requires no further relocations to its text. The code of a shared library should be validated using this assertion.

Run-Time Use of Shared Libraries

At run-time, `ld.so` finishes the job started by `ld`. That is, it performs the link-editing operations needed to resolve a program's remaining references using shared-library code and data. `ld.so`'s first task is to find and map in the required libraries. It uses slightly different search rules than `ld`. `ld.so` looks first in the directories specified by the *current* value of `LD_LIBRARY_PATH`, and then in the directories in the search path recorded by `ld` (the default directories and those specified by `-L`). In addition, `ld.so` attempts to find the "best" version of a shared library, that is, the version with the highest minor number (as described under *Version Control* below).

SunOS Shared Libraries

The shared libraries provided in SunOS are:

- The C library (both BSD and System V variants)
- Window libraries (`suntool` and `sunwindow`)
- `pixrect`
- kernel virtual memory access (`kvm`)
- The optional FORTRAN library (purchased and installed separately).

Static (`.a`) versions of these libraries are also provided.

Dynamic vs. Static Binding Semantics

There are some semantic differences between dynamic and static binding. These are not expected to cause a problem with programs that avoid questionable practices with regard to library search order. However, there is a potential for problems when programs are built from some components that have become dynamically loadable, while others remain static. Given the case where:

```
hermes% ld -o x ...dc sc
```

The executable `x` is composed of several objects, including a dynamic component, `dc`, and a static component, `sc`. `dc` was, prior to the introduction of shared libraries, an unordered archive file, and both `dc` and `sc` contain definitions for the symbol `getsym`. Suppose that `dc` contains a *reference* to `getsym`. If, in `dc`'s archive version, the definition for `getsym` preceded its reference, `ld` might have resolved that reference using the definition from `sc`. But in `dc`'s current (dynamic) form, its own definition is used instead. This is a result of the fact that at run-time, `ld.so` searches for a symbol definition starting with the main program, and then all `.so`'s in load order. Even though it allows for an inconsistency of this sort, this behavior preserves the ability to *interpose* definitions on library entry points.

Debuggers

The SunOS debuggers have been modified to deal with the dynamic linking environment provided by the new `ld`. In particular, they understand that symbol definitions may appear after a program starts executing. However debugger users must be aware that library symbols will not be resolved until `main()` has been called, as the next example shows.

```
hermes% cc -g -o test test.c
hermes% dbx test
Reading symbolic information...
Read 40 symbols
(dbx) stop in printf
no module, procedure or file named 'printf'
(dbx) stop in main
(1) stop in main
(dbx) run
Running: test
stopped in main at line 4 in file "test.c"
    4           printf("%d 0, errno);
(dbx) stop in printf
(3) stop in printf
(dbx) cont
stopped in printf at 0xed76954
0xed76954:                  moveml  #<d7,a5>,sp@
Current function is main
    4           printf("%d 0, errno);
```

Users of debugging tools also need to be aware that core files have incomplete information on the state of shared code. Core files contain only the stack and data regions of a process image. The text, and more importantly, the static data regions of dynamically loaded objects, do *not* appear. Thus, modifications made to initialized data are not reflected in the core file.

Performance Issues

Shared libraries represent a classic space vs. time trade-off. The work of incorporating the library code into an address space is deferred in order to save both primary and secondary storage. Therefore, one can expect to pay a slight CPU time penalty with programs that use shared libraries. This penalty can be attributed to added cost of:

- dynamically loading the libraries,
- performing the link editing operations, and
- the execution of the library PIC code.

However, these costs can be offset by the savings in I/O access time when library code is already mapped in by another program, since the (real) I/O time required to bring in a program and begin execution will be greatly reduced. As long as the CPU time required to merge the program and its libraries does not exceed the I/O time saved, the apparent performance of the program will be the same or better. However, if sharing does *not* occur, or if the system's CPU is already saturated, such savings may not be achieved.

Dependencies on Other Files

A dynamically bound program consists not only of the executable file that is the output of `ld`, but also of the files referred to during execution. Moving a dynamically bound program *may* also involve moving a number of other files as well. Moving (or deleting) a file on which a dynamically bound program depends may prevent that program from functioning.

Setuid Programs

For those programs that execute with an effective UID (user ID) or GID (group ID) different than the real UID or GID, `ld.so` ignores libraries in directories other than `/usr/lib`, `/usr/5lib` and `/usr/local/lib` in the search path.

1.3. Version Control

A version numbering mechanism has been provided for shared libraries. This allows newer compatible versions of a library to be bound at run-time. It also allows the link editors to distinguish between compatible and incompatible versions of a library.

Version Numbers of `.so`'s

The version number is composed of two parts, a *major* version, and a *minor* version number. This version-control suffix can be extended to an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the link editors at this time.

As noted earlier, `ld` records the version number of the shared library in the executable it builds. When `ld.so` searches for the library at run-time, it uses this number to decide which of the (possibly multiple) versions of a given library is "best," or whether *any* of the available versions are acceptable. The rules it follows are:

- **Major Versions Identical**: the major version used at execution time must exactly match the version found at `ld`-time. Failure to find an instance of the library with a matching major version will cause a diagnostic to be issued and the program's execution terminated.
- **Highest Minor Version**: in the presence of multiple instances of libraries that match the desired major version, `ld.so` will use the highest minor version it finds. However, if the highest minor version found at execution time is lower than the version noted at `ld`-time, a warning diagnostic is issued.

Major version numbers should be changed whenever there is an incompatible change to the library's interface.

NOTE *As always, the detection of incompatibilities between library versions remains the responsibility of the library's developer.*

Version Management Issues

Whenever there is an incompatible change to the library's calling interface, the major number of that library should be changed. A library's interface is defined by:

- the names and types of exported functions and their parameters; and
- the names and types of exported data (initialized or not)

Incompatible changes would include the deletion of a exported procedure, deletion of exported data, changes to an procedure's parameter list, and changes to data structures declared in a `.h` file normally included by both the library and the applications that use it.

Changes to internal library procedures and data do *not* constitute an interface change.

Minor versions should be changed to reflect *compatible* updates to libraries. An example of a compatible update would be changing a procedure's algorithm without changing its parameter list. Although adding a new library routine constitutes an interface change, it can be considered a compatible change.

Note that link-editors silently select the highest compatible version they can obtain. If the minor version used at `ld`-time is *higher* than the highest one found at run-time, then although the interfaces should remain compatible, it is possible that certain bug fixes or compatible enhancements on which the application depends might be missing: hence the warning message mentioned above.

1.4. Shared Library Mechanisms

There is no single mechanism in SunOS that implements shared libraries. Instead, the ability to construct a shared library comes as a consequence of enhancements to various existing facilities. The system components and their features that are instrumental in supporting shared libraries are:

- Virtual memory supports file mapping and "copy-on-write" sharing
- PIC generation by the compiler and assembler
- Link editor support for dynamic linking and loading

Memory Sharing

Memory sharing is provided by the kernel's virtual memory (VM) system. The mechanisms of interest for shared libraries are:

- File mapping by way of `mmap()`.
- Sharing at the granularity of a file page
- A per-page copy-on-write facility that allows run-time modification of a shared file, without affecting other users of that same file.

The VM system uses these features internally, so that an `exec()` of a program is reduced to establishing a copy-on-write mapping of the file containing the program. A *shared library* is added to the address space in exactly the same way, using this general file-mapping mechanism.

The C Compiler

The C compiler's `-pic` option generates position-independent code. When `-pic` is specified, references to objects that are external to the body of the code are made by way of *linkage tables*. These indirect references can degrade execution performance slightly, depending on of the number of dynamic references to global objects. The code sequences generated often assume that the linkage tables are no larger than a limit that is convenient for the specific machine (64K bytes for an MC68000, or 8K for a SPARC, for instance). In the (presumably rare) event the tables require a larger size, the compiler can be coerced into generating code sequences that permit larger linkage-table entries with the `-PIC` option.

Shared library code should be generated as PIC using either `-pic` or `-PIC` as appropriate. The use of PIC in shared libraries results in code that does not require relocation in order to be used, and is thus inherently sharable by any program that uses it. The same copy of PIC code can be shared among multiple programs, even if that code is placed at different addresses in each program. Any

dependence on actual addresses is isolated to the linkage tables, which are modified on a per- program basis to match the actual addresses selected.

The linkage tables are actually divided into two portions: a *Global Offset Table* (GOT) that provides indirections to data objects referenced by the PIC code, and a *Procedure Linkage Table* (PLT) that provides indirections to procedures referenced by the PIC code. The principal difference between the two types of indirections is that PLT entries are evaluated during dynamic linking, whereas GOT entries are evaluated at the start of execution.

The Assembler

Code generated by the `-pic` option requires support from the assembler. This support is enabled by the `-k` assembler flag, and is generated automatically by `cc` when invoking the assembler for a compilation performed with the `-pic` or the `-PIC` option.

User-written assembly code for use in a shared object must also be PIC. Refer to the appropriate *Sun-3 Assembly Language Reference* for your Sun system for details.

`crt0()`

Every main program produced by the standard languages is linked with a program prologue module, `crt0()`. This module contains the program's entry point, and performs various initializations of the environment prior to calling the program's `main()` function. `crt0()` refers to the symbol `__DYNAMIC`. As described above, when `ld` builds an executable requiring execution-time link editing, it defines this symbol as the address of a data structure containing information needed for execution-time link editing operations. If the structure is not needed, any reference to the symbol `__DYNAMIC` is relocated to zero.

At program start-up, `crt0()` tests to see whether or not the program being executed requires further link editing. If not, `crt0()` simply proceeds with the execution of the program as it always has – no further processing is involved. However, if `__DYNAMIC` is defined, `crt0()` opens the file `/usr/lib/ld.so` and requests the system to map it into the program's address space via the `mmap()` system call. It then calls `ld.so`, passing as an argument the address of its program's `__DYNAMIC` structure. `crt0()` assumes that `ld.so`'s entry point is the first location in its text. When the call to `ld.so` returns, the link editing operations required to begin the program's execution have been completed.

Link Editors: `ld` and `ld.so`

After `ld` has processed all of its input files, it attempts to resolve each symbolic reference to a relative offset within the executable being built. `ld` is able to complete this *symbolic* reduction at `ld`-time only if:

- all information relating to the program has been given and no `.so` will be added at execution time or
- the program has an entry point and symbolic reduction can be made for those symbols defined in the program

After performing all the reductions it can, if there are no further symbols to resolve, the output is a fully linked (static) executable. However, if any unresolved symbols remain, then the executable will require further link editing

at run-time. In this case, `ld` deposits the information (including version number) needed to obtain any needed `.so` files, in the data space of the incomplete executable.

It should be noted that uninitialized "common" areas (essentially all uninitialized C globals) are allocated by the link editor *after* it has collected all references. In particular, this allocation can not occur in a program that still requires the addition of information contained in a `.so` file, as the missing information may affect the allocation process. Initialized "commons," however, are allocated in the executable in which their definition appears.

After `ld` has performed all the symbolic reductions it can, it attempts to transform all relative references to absolute addresses. `ld` is able to do this *relative* reduction only if it has been provided some absolute address.

`ld.so`

At run-time, after receiving control from `crt0()`, `ld.so`, executes a short bootstrap routine that performs any relocations `ld.so` itself requires. It then processes the information contained in the `__DYNAMIC` structure of the program that called it. `ld.so` examines the list of required dynamic objects Each element of the list contains an offset relative to the `__DYNAMIC` structure of an array of `link_object` structures and has information to identify a `.so` that must be incorporated. The identification is the name specified on the `ld` command line used to build the program, and includes a bit indicating whether the object was named explicitly or via a `-l` option. Some version control information is also recorded for each entry in the `ld_need` array. `ld.so` looks up the indicated file, and maps it into the process's address space.

After all modules comprising the program have been placed in the address space, `ld.so` attempts to resolve the remaining symbols. After performing allocations for all uninitialized commons `ld.so` attempts to resolve all unbound references that occur *outside of procedure linkage tables.*

Unresolved procedural references in the linkage tables are not processed during program startup. Instead, such references are initialized such that the initial call results in a transfer of control to `ld.so`. When called in this way, `ld.so` first resolves the reference to an absolute address, and then modifies the linkage table entry to use that address. Deferring the binding of procedural entry-points until the first call eliminates unnecessary bindings to entry points that the program may not use.

Binding and Unbinding Routines: `dlopen()`, `dlsym()`, `dlclose()`, `dlerror()`

SunOS provides a programmatic interface to the run-time linker, which you can use to bind or unbind shared libraries during the course of program execution. `dlopen()` allows you to get access to a shared library, which it binds to the process's address space (if it isn't bound already). `dlsym()` returns the address of a given symbol within a (bound) shared library. `dlclose()` deletes a reference to a shared object. When the last reference is deleted, the shared object is removed from the process's address space. `dlerror()` can be used to obtain information about the last error occurring as the result of `dlopen()`, `dlsym()`, or `dlclose()`. Refer to `ld`(3) for details.

1.5. Building a Shared Library

In the simplest of cases, the commands needed to build a shared library might be:

```
hermes% cc -pic -c *.c
hermes% ld -o libx.so.1.1 -assert pure-text *.o
```

But note that this assumes that the library exports *no* initialized data. And it makes no guarantee that the library text makes the most efficient possible use of space, or allows for a minimal amount of paging.

As noted earlier, a shared library should be structured to avoid undue modification in the course of dynamic linking and execution. Otherwise, it is possible that some or all of the shared text may be rendered unsharable when run. Although this lack of sharing would not effect the *correct execution* of library routines, it will impact system performance. If only a few programs use the library, this impact is small. But for a widely-used library, the impact on system performance could be significant. Thus, shared library objects should be PIC, they should be validated using the `pure-text` assertion, and those libraries that export initialized data should be accompanied by a data interface description (`.sa`) file.

Building the `.so` File

To build the `.so` portion of a shared library, simply invoke `ld` with the list of object files that will comprise it. The version number is not automatically generated by `ld` (which creates a file named `a.out` by default), but you can specify the full name of the library, including the version number, with `ld`'s `-o` option. It is strongly suggested that you use the `-assert pure-text` assertion to uncover any instances of non-PIC code.

The `.sa` File

The `.sa` file is used to support `ld`'s `-dc` option, which provides a space/time efficient implementation of the interface between non-position-independent code and dynamically linked objects. The `.sa` file is an `ar`-format file (archive library) that contains the exported *initialized* data used by a shared library. When present, the `.sa` file it is statically linked at `ld`-time to insure correct allocation.

A data item is *exported* from a library if a program that uses the library refers to the data item *by name*. The contents of the data item are included if they are specified *by value* in the declaration. For instance, with a definition of the form:

```
char *strlist[] = { "string 1", "string 2" };
```

the data itself must be included in the `.sa` file, whereas with:

```
struct *strlist[] = { ptr1, ptr2 };
```

definitions for the objects named `ptr1` and `ptr2` would not *necessarily* have to be included. Note that if `ptr1` were itself defined as an initialized global in the library source, say:

```
extern char *ptr1 = NULL
```

then this definition would *also* have to go into the `.sa` file.

Uninitialized data (exported or not) is handled automatically, and need not be included in the `.sa` file. If the library does not export any data, then a `.sa`

would be unnecessary. The full name of a `.sa` also includes a version number that must match the version string of the `.so` it accompanies.

CAUTION **If a shared object exports initialized data, it is very important that a `.sa` file be created that contains such data. Failure to do so can degrade the performance of applications or, if the library is used heavily, the system as a whole. Further, in the event that such data is located within the text segment of the shared object, it is possible for `ld` to confuse the data with procedures defined by the library and to incorrectly link applications that reference such data.**

Initialized data can appear in the text segment of a shared object if it is part of a source file that is compiled with the `-R` (make initialized data read-only) option.

Building the `.sa` File

To build a `.sa` file:

1. Segregate the declarations of exported initialized data from the sources for each object, and place them in a separate source file. Make sure that an up-to-date object is compiled from each of those data-description sources, and include each of those data-description objects in both the static and shared versions of the library.
2. Create a separate (static) archive library composed of *only* the data-description objects, and give it a name of the form '`lib`*name*`.sa.`*version*'. This archive constitutes the `.sa` file. Be sure that the `.sa` has the same version number as the `.so` it is to accompany.
3. Use `ranlib`(1) to incorporate a symbol table within the `.sa` archive.

As an example, consider the system's C library. It contains a number of data structures that are initialized at program startup and which are exported to applications. Examples of these include the global variable `errno`, and the array of error messages `sys_errlist`.

The C library source has been constructed such that the variable `errno` appears in its own source file (`errno.c`). This accomplishes step 1 of the procedure outlined above. The relevant portion of this source file consists of the line:

```
int errno = 0;        /* global error return value, initially
```

This source file is compiled `-pic`, and the resulting object file, `errno.o`, is archived into the C library's `.sa` file. Since everything placed in a `.sa` file *must also* appear in the `.so` file. `errno.o` is also included in the `.so` file. Thus `errno.o` is also linked into the C library's `.so` file when it is built.

Once all such files have been placed in the `.sa` file, it is processed with `ranlib` to add a symbol table.

1.6. Building a Better Library

Library code that maximizes sharing is considered "better" because it makes more efficient use of the system's memory resources. Building the library components PIC is an important and easy first step, but there are other tuning strategies to consider as well.

Sizing Down the Data Segment

One way to maximize sharing is to minimize a `.so`'s data segment (containing initialized data), and its bss segment (containing uninitialized data). Often a `.so`'s data requirements are large because a significant portion of that data that is functionally read-only. There are several problems with this mix of read-only and modifiable data:

- data that could be shared is not,
- an unnecessary amount of swap space is reserved, and
- read-only data fragments the read-write storage, spreading it over more pages.

One approach is to move initialized read-only data into the text segment. This is done by compiling with the `-R` option. However caution needs to be exercised, since *initialized* data structures that contain pointers require relocation at run-time.

For instance, given the declarations:

```
void test();
int x;
struct fxy{
    void (*p0)();
    int *p1;
    };
struct fxy example = {test, &x};
```

The references to `&x` and `test` are instances of pointers embedded in an initialized structure. The actual addresses to which those pointers are resolved will not be determined until the program starts executing, and the shared object is placed in the address space. If this data structure is placed in the text segment of the shared object through the use of the `-R` option, then the relocation will cause that portion of the text segment to become unshared. Such data structures should not be contained in modules compiled with the `-R` option. You can check whether such relocations are occurring within a shared object by specifying the '`-assert pure-text`' option when building the shared object.

Using `xstr` to Extract String Definitions

Another common example of initialized data containing pointers is an array of strings:

```
char *errlist[] = {"err1", "err2"};
```

The `xstr(1)` utility can be used to make code containing initialized strings more sharable. It segregates the literal string data from its relocatable references, which allows the literal data to be merged safely into the text segment. However, files containing references to the string data should *not* be compiled with the `-R` option.

If there are several related pieces of data, another strategy is to coalesce the smaller items into a larger structure and allocate the space from the heap.

Better Ordering of Objects

The order of the objects in the executable can be important to minimizing the memory requirements. Since objects are concatenated together, linking in the wrong order may result in a unnecessarily large memory requirement. Two approaches that encourage better utilization of memory resources are:

- Routines that are frequently called should be packaged together, and isolated from startup or rarely-called code.
- A set of routines that represent a common sequence should also be packaged together. For example, given modules A, B, C, D, and E, where A and B fit on one VM page, C and D fit on another, and E fits on a partial page, if A always calls into E and never calls into B, the memory requirements may be reduced by a page if E follows A.

`crt0.o` Dependency

Sometimes a program will define its own `crt0()` initial routine. If it is intended that the program use shared libraries, then the programmer needs to provide a hook for the run-time linker. Further discussion of this can be found under `link`(5) in the *SunOS Reference Manual.*

The `ldconfig` Command

`ldconfig`(8) is a program used to construct a run-time linking cache for use by `ld.so`. The cache has a default list of directories `/usr/lib`, `/usr/5lib`, `/usr/lib/fsoft`, `/usr/lib/f68881`, `/usr/lib/ffpa`, and `/usr/lib/fswitch` and will accept as input a list of additional directories to augment this list. `ldconfig` records the pathname of the highest compatible version of each shared library in the specified search path.

At runtime, `ld.so` first queries the cache to determine which is the best version of a library in a particular directory. If the cache is unable to satisfy the request, `ld.so` enumerates the directory entries for the best version.

1.7. Shared Library Problems

`ld.so` Is Deleted

Since many system utilities are built to use shared libraries, and thus rely on dynamic link-editing, the potential exists for chaos if an important shared library (such as the C library) or `/usr/lib/ld.so` should be deleted.

If the latter has been deleted, you will see the following message:

```
crt0: no /usr/lib/ld.so
```

To deal with the chaos resulting from either the shared C library or `ld.so` being deleted, a number of commands and utilities have been statically linked. These include: `rcp`(1) `init`(8), `getty`(8), `sh`(1), `csh`(1), `mv`(1), `ln`(1), `tar`(1) and `restore`(8). Since most system utilities may be rendered unusable by this condition, it may be necessary to boot the system single-user in order to restore either `/usr/lib/ld.so` or the C library. Refer to *System and Network Administration* for procedures to restore these files.

Wrong Library Is Used

`ld.so` will not *detect* a library that is newly installed in the cache unless the cache is rebuilt using `ldconfig`. Thus, a program that depends on the newly-installed library may not be able to find it. You can use the `ldd`(1) command to identify the libraries on which a program depends.

Error Messages

```
ld.so: libname.so.major not found
```

`ld.so` failed to find a library with the appropriate major version number.

```
ld.so: open error for library
ld.so: can't read struct exec for library
ld.so: library is not for this machine type
```

Either the shared object has been corrupted, has incorrect access permissions, or was built to execute on another processor architecture.

```
ld.so: call to undefined procedure symbol from address
ld.so: Undefined symbol symbol
```

These messages generally indicate that the execution path attempts to refer to an undefined symbol. This is usually the result of a programming error.

```
ld.so.cache corrupted
```

The file `/etc/ld.so.cache` has become damaged. To correct it, remove the existing file and reboot the system. The file will be rebuilt.

```
ld.so: warning library has older version than expected
```

The version of the shared library that is currently being used has a minor version number that is lower than the version that was present at the time the application was compiled.

2

Lightweight Processes

2.1. Introduction

This tutorial provides some examples of how to use the lightweight process library. Although the term "lightweight processes" is often used, it is really a misnomer since the fundamental property of lightweight processes is not that they are somehow "lighter" than ordinary processes, but that a lightweight process represents a thread of control not bound to an address space. If threads appear to operate more efficiently than ordinary SunOS processes, it is because threads communicate via shared memory instead of a filesystem. Because threads can share a common address space, the cost of creating tasks and intertask communication is substantially less than the cost of using more "heavyweight" primitives. The availability of lightweight processes provides an abstraction well-suited to writing programs which react to asynchronous events such as servers. In addition, lightweight processes are useful for simulation programs which model concurrent situations.

Definition

The idea is to provide a process abstraction: a thread is a data type representing a flow of control. A number of operations are available to manipulate threads, including ways to control their scheduling and communication. Lightweight processes exist independently of virtual memory, I/O, resource allocation, and other operating system-supported objects, but are able to smoothly work with these objects.

The lightweight process abstraction for managing asynchrony is superior to the UNIX system signal abstraction. Under the UNIX system, a signal causes a sort of context switch (to a new instruction and optionally, to a new location on the stack) but the thread is the same: for example, you can `longjmp()` to the main program (the signal handler and main program can't run in parallel). Critical sections are implemented by disabling interrupts. With lightweight processes, the **only** way to manage an asynchronous activity is via a thread. There are no asynchronous exceptions in a thread. Critical sections are implemented with monitors. There is no need to lock out interrupts, with the concomitant possibility of losing information while in the critical section.

Functionality

The Sun *lightweight process library* provides primitives for manipulating threads, as well as for controlling all events (interrupts and traps) on a processor. The present library is supported for user-level processes only. This means that the time slice given to a process by the operating system is shared by all the threads within that process. Further, LWP objects are not accessible outside of

the containing process. Briefly, the primitives supported by the library include:

- Thread creation, destruction, status gathering, scheduling manipulation, suspend and resume
- Multiplexing the clock (any number of threads can sleep concurrently)
- Individualized context switching (e.g., it is possible to specify that a given set of threads will touch floating point registers and only those threads will context switch these registers)
- Monitors and condition variables to synchronize threads
- Extended rendezvous (message send-receive-reply) between threads
- An exception handling facility that provides both *notify* and *escape* exceptions
- A way to map interrupts into extended rendezvous
- A way to map traps into exceptions
- Utilities to allocate red-zone-protected stacks, and to provide some stack integrity checking for environments that lack sophisticated memory management

Scheduling is by default, priority-based, non-preemptive within a priority. However, sufficient primitives are available that it is possible to write your own scheduler. For example, to provide a round-robin time-sliced scheduler, a high-priority thread may periodically reshuffle the queue of time-sliced threads which are at a lower priority. Although pure coroutine scheduling is possible, it is *not* required and purely preemptive scheduling may be used. Threads currently lack kernel support, so system calls still serialize thread activity, although the *non-blocking I/O library* (*libnbio.a*) mitigates this problem somewhat. When a set of threads are running, it is assumed that they all share memory.

Tutorial Goals

This tutorial provides some practical examples of how to program using lightweight processes. Also included is some discussion of the rationale for the lightweight process primitives. Syntax details of the lightweight process primitives are not supplied in this tutorial, though they can be found in the *SunOS Reference Manual.*

2.2. Threads

The lightweight process mechanism allows several *threads* of control to share the same address space. Each lightweight process is represented by a procedure which will be converted into a thread by the `lwp_create()` primitive. Once created, a thread is an independent entity, with its own stack as supplied by its creator. `lwp_create()` performs a number of actions: a thread context is allocated, the stack is initialized, and the thread is made eligible to run. A collection of threads runs within a single ordinary process. This collection is sometimes called a *pod.*

Lightweight processes (*LWP's* or *threads*) are scheduled by priority. It is always the case that the highest priority non-blocked thread is executing. Threads may block on certain occurrences, such as the arrival of a message or the procurement

of a monitor lock. Within a priority, threads execute on a first-come, first-served basis. Thus, if two threads are created at the same priority, they will execute in the order of creation.

Here is an example of how to do something simple with lightweight processes. The program below creates a thread which prints out the "hello world" message and then terminates (by "falling through" the procedure). `main()` becomes a lightweight process as soon as a LWP primitive (here, `pod_setmaxpri()`) is called. Note that `main()` is created with a priority of `MAXPRIO` so that it may set things up as it wishes before allowing other threads to run.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    thread_t tid;
    int task();

    printf("main here\n");        /* 1 */
    (void)pod_setmaxpri(MAXPRIO);    /* 2 */
    lwp_setstkcache(1000, 2);    /* 3 */
    lwp_create(&tid, task, MAXPRIO,
        0, lwp_newstk(), 0);     /* 4 */
}

task()
{
    printf("hello world\n");
    /* now, fall through and terminate this thread */
}
```

The command to compile this program (call it foo.c) is:

```
example cc -o foo foo.c -llwp
```

Let's go through this program line by line. We begin by printing a message "main here" at line 1. Then, `pod_setmaxpri()` turns `main()` into a lightweight process (as it's the first LWP primitive to be called). `pod_setmaxpri()` also specifies the maximum scheduling priority: in this case, 10. The range of scheduling priorities 1..10 is now available to the client. If we didn't use `pod_setmaxpri()` the available priority would be just MINPRIO. Now, `main()` is a thread running at a priority of 10, the maximum priority. In other words, `main()` will execute until it explicitly blocks or otherwise yields control to another thread.

`lwp_setstkcache()` initializes a cache of stacks that can be used by subsequent `lwp_newstk()` calls. `lwp_newstk()` will return a stack of *at least* the size specified in the `lwp_setstkcache()` call (here, 1000 bytes), and this stack is red-zone protected. The second argument to `lwp_setstkcache()`

specifies how big the cache should be initially (how many stacks it should contain). Larger numbers will require more memory, but will make cache faults less likely. On a fault, an additional cache of the same size will be allocated. A stack allocated from the stack cache will automatically be freed when the thread that uses it dies. Allocation from this cache is almost as efficient as using statically allocated stacks.

At line 4, we create a new thread. This thread will begin execution at `task()`, have a scheduling priority of 10, use the stack cache for a stack, and take no arguments initially. Even though it will run at the same priority as `main()`, `task()` will not run until `main()` relinquishes control because of the FCFS scheduling policy for threads at the same priority, and `task()` is at the same priority as `main()`. (It is not a good programming practice to rely on the ordering of threads within a priority since this assumption may not hold on a multiprocessor or in the presence of external scheduling). The identity of the new thread is returned in `tid`. This identity may be used in subsequent LWP primitives.

When the `main()` thread "falls through", it terminates. At this point, `task()` will run, print its message, and terminate. The LWP library will notice that no more threads remain, and the program will terminate.

Be careful not to confuse threads with ordinary heavyweight processes. For example, there are no inheritance rules about lightweight processes, and lightweight processes do not have their own set of descriptors.

Stack Issues

Stack Size

A major problem is to determine how big to make the thread stacks. Once this determination is made, you can decide how or if you need protection against exceeding this limit. UNIX presents the same problem to the user, but it rarely causes trouble because the maximum stack length is very big. Allocating large stacks is not a big performance drain because pages are only allocated if actually used. Hence, you can allocate very large stacks fairly casually.

Protecting Against Stack Overflow

`lwp_newstk()` automatically allocates red-zone protected stacks (references beyond the stack limit will generate a SIGSEGV event). There are two ways to ensure stack integrity when not using `lwp_newstk()`. One way is to use the `CHECK()` macro at the beginning of each procedure (before any locals are assigned), in conjunction with the `lwp_checkstkset()` primitive. If the procedure exceeds the thread stack limit, the procedure will return and set a global variable. Another way is to use the `lwp_stkcswset()` primitive. This enables stack checking on context switching. Although this is transparent to the client programs, it may not detect errors until after the stack limit has been exceeded. Thus, with `lwp_stkcswset()`, an error is considered fatal. `CHECK()` detects errors before any damage is done, so error recovery is possible.

It is possible to assign a statically allocated stack to a thread. Thus, in the program above, we could declare a stack as follows, using the macros defined in

`stackdep.h` to declare the stack portably. `MINSTACKSZ()` is added to include any stack room needed by the LWP library to execute the LWP primitives.

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <lwp/stackdep.h>

#define MINSTACKSZ 1024
#define MAXPRIO 10

stkalign_t stack[1000+MINSTACKSZ];

main()
{
    int task();
    thread_t tid;

    (void)pod_setmaxpri(MAXPRIO);
    lwp_create(&tid, task, MAXPRIO, 0, STKTOP(stack), 0);
}

task()
{
    printf("task: hello world\n");
}
```

Coroutines

It is possible to use threads as pure coroutines in which one thread explicitly yields control to another. `lwp_yield()` allows a thread to yield to either a specific thread at the same priority, or the next thread in line at the same priority. Here is an example of three coroutines: `main()`, `coroutine()`, and `other()`. The result should be the numbers 1 through 7 printed in sequence. In the case where a generic yield is done (`lwp_yield(THREADNULL)`), the current thread goes to the end of its scheduling queue. When a specific yield is done, the specified thread butts in front of the current one at the front of the scheduling queue. Since we are just using coroutines, a single priority (MINPRIO) is sufficient and we do not increase the number of available priorities with `pod_setmaxpri()`.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t co1;        /* main's tid */
thread_t co2;        /* coroutine's tid */
thread_t co3;        /* other's tid */
main(argc, argv)
    int argc;
    char **argv;
{
    int coroutine(), other();
    lwp_self(&co1);

    lwp_setstkcache(1000, 3);
```

```
    lwp_create(&co2, coroutine, MINPRIO, 0,
      lwp_newstk(), 0);
    lwp_create(&co3, other, MINPRIO, 0, lwp_newstk(), 0);
    printf("1\n");
    lwp_yield(THREADNULL);   /* yield to coroutine */
    printf("4\n");
    lwp_yield(co3); /* yield to other */
    printf("6\n");
    exit(0);
}

coroutine() {
    printf("2\n");
    if (lwp_yield(THREADNULL) < 0) {
        lwp_perror("bad yield");
        return;
    }
    printf("7\n");
}

other() {
    printf("3\n");
    lwp_yield(THREADNULL);
    printf("5\n");
}
```

Custom Schedulers

There are three ways to provide scheduling control of threads to the client. One way is to do nothing and simply provide the client a pointer to a thread context which can be scheduled at will. This method suffers from the fact that most clients don't want to be bothered by constructing their own scheduler from scratch. Another way to do it is to provide a single scheduling policy, with very little client control over what runs next. The UNIX system provides such a policy. While this is the simplest (from the point of view of the client) way to go, it makes it difficult to implement policies that take into account the differing response time needs of client threads. We chose to take a middle ground in an effort to avoid these problems. There is a default scheduling policy, but enough primitives are provided that it is possible to construct a wide variety of scheduling policies based on it.

It is possible to custom-build your own scheduler by using the primitives `lwp_suspend()`, `lwp_yield()`, `lwp_resume()`, `lwp_setpri()`, and `lwp_resched()`. `lwp_suspend()` may also be used in debugging, to ensure that a thread is stopped before inspecting it. Here, we give an example of how to build a round-robin time-sliced scheduler. The idea is to have a high priority thread act as a scheduler, with the other threads at a lower priority. This scheduler thread simply sleeps for the desired quantum. When the quantum expires, the scheduler issues a `lwp_resched()` command for the priority of the scheduled threads. This causes a reshuffling of the run queue at that priority.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
```

```
#define MAXPRIO 10
main(argc, argv)
    int argc;
    char **argv;
{
    int scheduler(), task(), i;
    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 5);
    (void) lwp_create((thread_t *)0, scheduler, MAXPRIO, 0,
      lwp_newstk(), 0);
    for (i = 0; i < 3; i++)
        (void) lwp_create((thread_t *)0, task, MINPRIO, 0,
          lwp_newstk(), 1, i);
    exit(0);
}

scheduler() {
    struct timeval quantum;
    quantum.tv_sec = 0;
    quantum.tv_usec = 10000;
    for(;;) {
        lwp_sleep(&quantum);
        lwp_resched(MINPRIO);
    }
}

/* these tasks are scheduled round-robin, preemptive */
task(arg) {
    for(;;)
        printf("task %d\n", arg);
}
```

Special Context Switching

A thread can pretend to be the only activity executing on its machine even though many threads are running. The LWP library is the entity that provides this illusion. As such, the LWP library provides for *context switches* between threads which cause volatile machine resources to be multiplexed so that each thread operates with its own set of machine resources. In many cases, a context switch requires only that machine registers and the stack be multiplexed. In other cases, floating point state, memory management registers, and even software state may be multiplexed as well. The LWP library allows threads to have differing amounts of switchable state to efficiently allow processes with different resource needs to coexist.

In addition to switchable state, a thread will possess state that is updated by other primitives. This per-thread state includes such information as messages sent to a thread, and monitor locks it holds. The only per-thread state maintained by the library is that used to support the LWP primitives, whereas heavyweight processes entail a considerable amount of per-process state. With threads, this amount of state is much smaller with the intent that only those threads which need to should maintain additional state. Thus, operating-system-specific information such as signal state, accounting information, and file descriptors is not

found in the thread context. It is up to the clients to provide as much "weight" as is required.

The reason that special contexts are not directly incorporated into the context of a thread is that not all threads will use these contexts and there is no reason to make a thread pay for something it won't use. The LWP library will allocate a new context buffer for each special context a thread is initialized with, and pass a pointer to this context to the save and restore routines defined for this context. The id of the previous and new threads to use the context are also passed in, in case the save and restore routines maintain per-thread information about a special context. This information could be used, for example, by a memory-management special context to avoid doing work if the previous and current threads access the exact same memory management registers.

To use the special context mechanism, you first define a special context with the `lwp_ctxset()` primitive. This requires that you figure out how to save and restore the state required by your context and provide procedures to do this. In the example below, which context-switches the C-library global `errno`, the routines `__libc_save()` and `__libc_restore()` are provided, and the context they will save into and restore from is of type `libc_ctxt_t`. The routine `libcenable()` is used to define the context, and the global `LibcCtx` remembers the cookie that defines the context.

Once a special context is defined, you may initialize any thread to use the resource multiplexed by the special context by using `lwp_ctxinit()`. The initialization of a given thread to use a special context can be done directly, or, if the resource permits, by catching a trap when the resource is first used by a thread. In the example below, we expect that each thread accessing `errno` will be initialized via `libcset()` to use the special libc context. Threads protected with this special context can read `errno` without fear that another thread can change `errno` (e.g., via a system call) from underneath them. Because this `errno` multiplexing is quite useful, it is available in the routine `lwp_libcset()` which does all of the work for you.

```
#include <lwp/lwp.h>

#define TRUE 1

typedef struct libc_ctxt_t {
    int libc_errno;
} libc_ctxt_t;
static int LibcCtx;

/* enable libc special contexts */
libcenable()
{
    extern void __libc_save();
    extern void __libc_restore();

    LibcCtx = lwp_ctxset(__libc_save, __libc_restore,
        sizeof (libc_ctxt_t), TRUE);
}

/* set a thread to have libc context */
```

```
lwp_libcset(tid)
    thread_t tid;
{
    (void) lwp_ctxinit(tid, LibcCtx);
}

/* routines for saving/restoring global library data. */
void
__libc_save(cntxt, old, new)
    caddr_t cntxt;
    thread_t old;
    thread_t new;
{
    extern int errno;
#ifdef lint
    old = old;
    new = new;
#endif lint

    ((libc_ctxt_t *)cntxt)->libc_errno = errno;
}

void
__libc_restore(cntxt, old, new)
    caddr_t cntxt;
    thread_t old;
    thread_t new;
{
    extern int errno;
#ifdef lint
    old = old;
    new = new;
#endif lint

    errno = ((libc_ctxt_t *)cntxt)->libc_errno;
}
```

2.3. Messages

Messages vs. Monitors

There are two predominant types of process synchronization in use today: the rendezvous paradigm and the monitor paradigm. The lightweight process package provides both, in part to avoid denying a large number of people their favorite primitives, and in part because each has compelling reasons.

Rendezvous has the advantages that it maps cleanly to Sun interprocess-communications facilities (Sun RPC), can potentially support communication across different address spaces, is higher-level than monitors because both data transmission and synchronization are combined into a single concept, and is a natural way to map asynchronous events into higher-level abstractions since messages are reliable and conditions are not.

The big advantage with monitors are their familiarity to UNIX system programmers (via similarity to `sleep()` and `wakeup()` in the kernel), and the efficiency win when protected data is accessed: with rendezvous, a context switch is always required; with monitors, a context switch is only necessary if the monitor lock is busy at the time of access.

Rendezvous Semantics

To use messages, one thread issues a `msg_send()` and another thread issues a `msg_recv()`. Whichever thread gets to the corresponding primitive first waits for the other, hence the term *rendezvous*. When the rendezvous takes place, the sender remains blocked until the receiver decides to issue a `msg_reply()`. Immediately after `msg_reply()` returns, both threads are unblocked.

It is the responsibility of the sender to provide the buffer space both for a message to be sent to the receiver, and for a reply message from the receiver. Either of these messages may be empty. While the sender is blocked, the receiver has access to the buffers provided by the sender. When the receiver replies, she is undertaking not to use these buffers any more: the transaction is complete. If memory management was used to share address spaces, the sender's buffers would be mapped into the receiver's address space only for the duration of the rendezvous. Because both send and receive buffers are provided by the sender, there is no need for further synchronization to tell the receiver that her reply was accepted by the sender.

Sometimes it is desired to perform a *non-blocking* send in which the sender does not block on a send request. We did not provide this as a primitive because it is easily implemented by using an additional thread to do the send.

Messages and Threads

Messages are sent to threads, and each thread has exactly one queue associated with it to receive messages on. We could have provided message queues (ports) as objects not bound to processes. This would give more flexibility, but would require a more complex selection primitive to really justify the extra functionality. In addition, it would complicate the implementation because we desire to terminate a rendezvous on behalf of the remaining thread should one of the rendezvousing threads be destroyed.

To receive a rendezvous request, a process specifies the identity of the sending thread it wishes to rendezvous with. Optionally, a receiver may specify that *any* sender will do. There is no other form of selection available. If more power is needed, the client can build server processes which act as intelligent ports capable of performing complex selection criteria. Note that the id of the sending thread or agent is supplied to the receiver by the LWP library, so that it is not possible to forge the identity of the sender.

Here is an example of basic message passing. `main()` creates two threads, `sender()` and `receiver()`. Because it has a higher priority, the receiver starts first and blocks, awaiting a rendezvous. Then, the sender runs and prepares a message. However, the sender sleeps for 2 seconds before sending it. In this time, the receiver gave up waiting and tried again, now waiting with infinite patience. The sender wakes up a second later and attempts to rendezvous with the receiver. This rendezvous immediately succeeds, the receiver reads the message, prepares a reply, and replies. At this point, the rendezvous is complete and both

sender and receiver are runnable processes. Because the receiver has a higher priority, the message "done receiving" is printed ahead of the "got reply" message. Note that the receiver should *not* touch any of the data mentioned in the send once the reply has been made.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#define MAXPRIO 10

thread_t c1, c2;

main(argc, argv)
    int argc;
    char **argv;
{
    int sender(), receiver();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);
    lwp_create(&c1, sender, MINPRIO, 0, lwp_newstk(), 0);
    lwp_create(&c2, receiver, MINPRIO+1, 0,
      lwp_newstk(), 0);
    exit(0);
}

sender() {
    char out[20];
    char in[30];
    int i;
    struct timeval wait;

    wait.tv_sec = 2;
    wait.tv_usec = 0;

    for (i = 0; i < 19; i++)
        out[i] = (int)'A' + i;
    out[19] = '\0';
    lwp_sleep(&wait);
    if (msg_send(c2, out, 20, in, 26) == -1) {
        lwp_perror("msg_send");
        return;
    }
    printf("got reply %s\n", in);
}

receiver() {
    int i;
    struct timeval wait;
    char *arg, *res;
    int asz, rsz;
    thread_t sender;

    wait.tv_sec = 1;
    wait.tv_usec = 0;

    /* try one second */
```

```
    sender = THREADNULL;        /* take message from anyone */
    if (msg_recv(&sender, &arg, &asz, &res, &rsz, &wait)
      == -1) {
        if (lwp_geterr() != LE_TIMEOUT) {
            lwp_perror("msg_recv");
            return;
        }
        /* wait forever or until message arrives from sender */
        if (msg_recv(&sender, &arg, &asz, &res, &rsz,
          INFINITY) == -1) {
            lwp_perror("msg_recv");
            return;
        }
    }
    printf("got message %s\n", arg);
    for (i = 0; i < rsz - 1; i++)
        res[i] = (int)'B' + i;
    res[rsz - 1] = '\0';
    msg_reply(sender);
    printf("done receiving\n");
}
```

Intelligent Servers

Because the reply can be done at any time, a receiver can receive a number of messages before replying to them. This makes it possible to implement complex servers. In the following example, processes send requests in a random order to a server thread. The server serializes the requests and processes them in the order associated with the request.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
thread_t pt;

typedef struct port_msg {
    int order;
    char *msg;
} port_msg;

#define MAXPRIO 10
main(argc, argv)
    int argc;
    char **argv;
{
    int process();
    int port();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);

    /* argument to new thread is order # */
    lwp_create((thread_t *)0, process, MINPRIO, 0,
      lwp_newstk(), 1, 3);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
```

```
        lwp_newstk(), 1, 0);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
        lwp_newstk(), 1, 2);
    lwp_create((thread_t *)0, process, MINPRIO, 0,
        lwp_newstk(), 1, 1);
    lwp_create(&pt, port, MAXPRIO, 0, lwp_newstk(), 0);
    exit(0);
}

process(id)
    int id;
{
    port_msg m;
    char buf[10];

    m.order = id;
    m.msg = buf;
    printf("sending %d\n", id);
    msg_send(pt, (char *)&m, sizeof(port_msg), 0, 0);
    printf("%d replied to\n", id);
}

/*
 * collect messages in any order, process them in order
 */
port()
{
    thread_t sender;
    char *arg;
    int asz;
    port_msg *request;
    thread_t senders[4];
    int i;

    for(i = 0; i < 4; i++) {
        /* convenient way to receive from any sender */
        MSG_RECVALL(&sender, &arg, &asz, 0, 0, INFINITY);
        request = (port_msg *)arg;
        printf("got %d\n", request->order);
        senders[request->order] = sender;
    }
    for (i = 0; i < 4; i++) {
        msg_reply(senders[i]);
    }
}
```

2.4. Agents

Some environments will present asynchronous interrupts to the client. For example, on a bare machine, a character typed at a tty can cause an interrupt to randomly steal control away from the executing program. Similarly, a signal can interrupt the current thread. Because of the random nature of interrupts, it is hard to understand programs that deal with them. The lightweight process library provides a simple way to transform asynchronous events into synchronous ones.

A message paradigm (as opposed to a monitor paradigm) was chosen for mapping interrupts because an interrupt cannot wait for a monitor lock if held by a client. Even if condition variables are used outside of a monitor, it is still necessary to add memory to the condition variable to prevent races (just before the client decides to sleep, an interrupt comes in, causing a condition to be notified, which is missed by the client, who then sleeps, resulting in deadlock). Adding a flag to a condition to prevent this is analogous to converting the condition into a 1-bit message.

With asynchronous interrupts, an event causes a sort of context switch within the *same* thread. With LWP's, a thread must synchronously *rendezvous* with an interrupt. Thus, to have an event do something asynchronously, it is necessary to use a separate thread to handle it. To simulate typical UNIX signal handling, you would create two threads: one thread to represent the main program, and another thread at a higher priority to represent the signal handler. The latter thread would have an *agent* set up to receive signals.

The agent mechanism is provided to map asynchronous events into messages to a lightweight process. A message from an agent looks exactly like a message from another thread. When you create an agent, you also provide a portion of the pod's address space for the agent to store its message. You cannot receive the next message from an agent until you reply to the current one. Because the LWP scheduler is preemptive, when a signal is mapped into a message, it will cause the highest priority thread blocked on the agent to run next. Client threads which have agents can use all of the LWP library facilities (monitors, condition variables, messages) to synchronize with other threads.

The agent mechanism does its best to process signals as rapidly as possible. Nonetheless, it is possible that events will be missed because the kernel does not remember more than one signal occurring while a signal is being processed. Furthermore, signals are not delivered for each occurrence of I/O. Therefore, a thread which wakes up from a `SIGIO` agent for example, should not sleep again until `read()` on the descriptor fails, indicating that another SIGIO will be delivered when more I/O is available.

When an interrupt arrives, the LWP library saves only volatile information about the interrupt, and wakes up any threads waiting on the agent. On a bare machine, volatile information would include for example, the character typed in from a tty. Under SunOS, volatile information includes the state normally delivered to a signal handler as well as the identity of the thread running at the time of the event. This volatile information is passed as a message to the client thread.

System Calls

Non-blocking I/O Library

A set of heavyweight processes can execute concurrently in the kernel. For example, three heavyweight processes can concurrently initiate writes to the same device. This is not the case for lightweight threads. Some relief can be provided by marking descriptors asynchronous with `fcntl(2)`. This allows threads to block on `SIGIO` agents and only block on a system call when it is likely to be immediately productive (i.e., without blocking indefinitely).

Similarly, a thread can block on a `SIGCHLD` agent instead of blocking on a `wait(2)` system call. However, there is no general solution to the problem of having several threads execute system calls concurrently until the LWP primitives are made available as true system calls operating on a shared set of descriptors. The use of the non-blocking I/O library can help by automatically blocking a thread attempting any I/O until such I/O is likely to succeed immediately. The blocked thread will try the system call again automatically when a SIGIO event occurs.

Using the Non-Blocking IO Library

Here is an example of how to use the non-blocking IO library. We have a procedure *compute_pi* that runs at low priority, and a procedure *reader* that runs at high priority. If we link this program without the non-blocking IO library, the reader will prevent the compute-bound thread from running since the `read()` system call blocks. However, if we link in the non-blocking IO library, the compute-bound procedure will execute until some IO is made available (in this case, by the user typing something at the terminal).

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    int reader();
    thread_t tid;

    pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 2);
    lwp_create(&tid, reader, MAXPRIO, 0, lwp_newstk(), 0);
    lwp_setpri(SELF, MINPRIO);
    compute_pi();
    exit(0);
}

reader()
{
    char buf[256];
    int cnt;

    for(;;) {
        cnt = read(0, buf, 256);
        buf[cnt] = 0;
        printf("\ngot %s\n", buf);
    }
}

compute_pi()
{
    for(;;) {
        /* compute pi to a zillion places */
    }
}
```

Here is another example of how to use the non-blocking I/O library. The first program is a server which accepts requests over the wire. When a request arrives, a thread is created to handle the request so that accepting and processing the requests can proceed in parallel. The processing of the request consists in sleeping for the amount of time specified in the request message. Note that if the non-blocking I/O library is not linked in, the main program loop prevents any (lower priority) request-processing threads from executing. `lwp_datastk()` is used to put the message on the stack of the newly-created thread. Thus, there is no need to keep the message in *main*.

```
/*
 * sleep server program.
 */
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>

#define MYPORT  8889
#define MAXPRIO 10
#define BUFSIZE 10

struct message {
    int timeout;
    int msgsize;
    char buf[BUFSIZE];
} message;
extern int errno;

main()
{
    int s;
    struct sockaddr_in addr;
    int len = sizeof(struct sockaddr_in);
    int fromlen;
    int rlen;
    void compute();
    stkalign_t sp;
    caddr_t loc;

    if (pod_setmaxpri(MAXPRIO) < 0) {
        lwp_perror("pod_setmaxpri");
        _exit(1);
    }
    if (lwp_setstkcache(5000, 5) < 0) {
        lwp_perror("lwp_setstkcache");
        _exit(1);
    }
    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
```

```
        perror("can't get socket");
        _exit(1);
    }
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_family = AF_INET;
    addr.sin_port = MYPORT;
    if (bind(s, (struct sockaddr *)&addr, len) < 0) {
        perror("bind");
        close(s);
        _exit(1);
    }
    if (getsockname(s, (caddr_t)&addr, &len) != 0) {
        perror("can't get name");
        close(s);
        _exit(1);
    }
    for(;;) {
        do {
            fromlen = len;
            rlen = recvfrom(s, (caddr_t)&message,
              sizeof(struct message), 0,
              &addr, &fromlen);
        } while ((rlen == -1) && (errno == EINTR));
        if (rlen == -1) {
            perror("recvfrom");
            _exit(1);
        }
        sp = lwp_datastk(message.buf,
          message.msgsize, &loc);
        lwp_create((thread_t *)0, compute, MINPRIO,
          0, sp, 2, message.timeout, loc);
    }
    exit(0);
}

void
compute(timeout, msg)
    int timeout;
    char *msg;
{
    struct timeval time;
    time.tv_sec = timeout;
    time.tv_usec = 0;

    printf("%s\n", msg);
    lwp_sleep(&time);
    printf("%s slept %d secs\n", msg, timeout);
}
/*
 * program to send a message to the sleep-server.
 * usage: slp <servername> <timeout in seconds> <message>
 */
#include <sys/types.h>
#include <netinet/in.h>
```

```
#include <sys/socket.h>
#include <netdb.h>
#include <errno.h>

#define MYPORT  8889
#define BUFSIZE 10

struct messsage {
    int timeout;
    int msgsize;
    char buf[BUFSIZE];
} message;

extern int errno;

main(argc, argv)
    int argc;
    char **argv;
{
    int s;
    struct sockaddr_in addr;
    int len = sizeof(struct sockaddr_in);
    int err;
    struct hostent *hp;
    char *server;

    if (argc != 4) {
        printf("usage: %s server seconds message\n",
          argv[0]);
        exit(2);
    }
    server = argv[1];
    message.timeout = atoi(argv[2]);
    message.msgsize = strlen(argv[3]) + 1;
    bcopy(argv[3], message.buf, message.msgsize);
    if ((hp = gethostbyname(server)) == 0) {
        printf("can't get host name\n");
        exit(1);
    }
    bcopy(hp->h_addr, &addr.sin_addr, hp->h_length);
    addr.sin_family = AF_INET;
    addr.sin_port = MYPORT;

    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
        perror("can't get socket");
        exit(1);
    }

    do {
        err = sendto(s, (caddr_t)&message,
          sizeof(message), 0, &addr, len);
    } while ((err == -1) && (errno == EINTR));
    if (err == -1) {
        perror("sendto");
        exit(1);
    }
```

```
    exit(0);
}
```

A final example of the non-blocking I/O library illustrates how the `wait(2)` system call can be used. Here, the parent process forks two children. The children do something (in this case, they just sleep) and terminate with an exit status. The parent would like to reap the children, but does not want to block in the process. The solution is to link in the non-blocking I/O library which lets the parent block without stopping other threads. Behind the scenes, a SIGCHLD agent thread is watching for terminating processes. If the non-blocking I/O library is not linked in, the *wait* will succeed, but the *otherwork* thread will not get a chance to run. Note that threads using system calls remapped by the non-blocking I/O library automatically receive the C-library special context, so *errno* is not lost across context switches.

```
#include <lwp/lwp.h>
#include <lwp/lwpmachdep.h>
#include <signal.h>

main()
{
    int child;
    union wait stat;
    void otherwork();

    (void)pod_setmaxpri(10);
    (void)lwp_setstkcache(1000, 2);
    (void)lwp_create((thread_t *)0, otherwork,
      MINPRIO, 0, lwp_newstk(), 0);
    if (fork() == 0) {
        sleep(5);
        _exit(7);
    } else if (fork() == 0) {
        sleep(3);
        _exit(5);
    }
    for (;;) {  /* reap children */
        child = wait(&stat);
        printf("%d got %d\n", child, stat.w_retcode);
        if (child == -1) {
            perror("wait");
            break;
        }
    }
    exit(0);
}

void
otherwork()
{
    struct timeval time;
    time.tv_sec = 2;
```

```
    time.tv_usec = 0;
    for(;;) {
        printf("otherwork here\n");
        lwp_sleep(&time);
    }
}
```

Examples of Agents

We present two examples of agent use below. The first example shows how a traditional UNIX signal handler can be emulated. Note the use of monitors to protect access to shared state. The second example shows the use of a SIGIO agent.

```
/* Example of the UNIX system style of signal handling */
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <signal.h>

#define MAXPRIO 10
mon_t mid;
int shared_state;

main(argc, argv)
    int argc;
    char **argv;
{
    int sigint_catch();
    int task();
    int task1();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    mon_create(&mid);
    (void) lwp_create((thread_t *)0, sigint_catch, MAXPRIO,
      0, lwp_newstk(), 0);

    /*
     * the signal handler will preempt the main program
     * so we give it the higher priority
     */
    lwp_setpri(SELF, MINPRIO);
    for(;;) {
        /* do other work */;
        mon_enter(mid);
        /* access shared_state */
        mon_exit(mid);
    }
    exit(0);
}

sigint_catch()
{
    eventinfo_t sigmem;
    char *arg;
```

```
    int asz;
    thread_t sender;

    agt_create(&sender, SIGINT, (char *)&sigmem);
    for(;;) {
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        printf("got ^C\n");
        mon_enter(mid);
        /* access shared_state */
        mon_exit(mid);

    }
}

/* Example showing how to process SIGIO */

/*
 * Some points about this code:
 * 1. because the system call could be interrupted, we
 * check for EINTR. In order that errno is accurate, we
 * make sigio_catch a libc thread (else, it may be lost
 * on a context switch).
 *
 * 2. We reset stdin before returning so the shell won't
 * get confused. (It would otherwise get EWOULDBLOCK
 * trying to read stdin, and bomb out with an error).
 */

#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#define TRUE 1
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
{
    int sigio_catch();
    thread_t tid;

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    lwp_create(&tid, sigio_catch, MAXPRIO,
      0, lwp_newstk(), 0);
    lwp_libcset(tid);
    lwp_setpri(SELF, MINPRIO);
    /* do main's work */
}

sigio_catch()
{
    int cnt;
```

```
    char buf[256];
    int fd = 0; /* stdin */
    extern int errno;
    int emask, rmask, wmask;
    eventinfo_t agtmemory;
    thread_t sender;
    char *arg;
    int asz;
    int inputbits = 01 << fd;

    /*
     * Enable SIGIO on stdin.  When we actually read, it
     * may still return EWOULDBLOCK (SIGINT before SIGIO
     * delivered flushes input leaving nothing to read),
     * so need to read again.
     */
    fcntl(fd, F_SETFL, FASYNC|FNDELAY);
    rmask = inputbits;
    emask = wmask = 0;
    agt_create(&sender, SIGIO, &agtmemory);

    for(;;) {

        /*
         * block pending notification that reading would
         * be useful meanwhile, main can get work done.
         */
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        select(32, &rmask, &wmask, &emask,
          (struct timevel *)0);
        if (rmask & inputbits) {
            cnt = read(fd, buf, 256);
            if (cnt != -1 || errno != EWOULDBLOCK ||
              errno != EINTR)
                break;
        }
    }
    buf[cnt] = 0;
    printf("\ngot %s\n", buf);
    fcntl(fd, F_SETFL, 0); /* reset stdin so no
                  shell confusion */
}

/*
 * To do simple signal handling within main,
 * we could just write:
 */

main(argc, argv)
    int argc;
    char **argv;
{
    int cnt;
    char buf[256];
```

```
    int fd = 0; /* stdin */
    extern int errno;
    int emask, rmask, wmask;
    eventinfo_t agtmemory;
    thread_t sender;
    char *arg;
    int asz;
    int inputbits = 01 << fd;

    (void)pod_setmaxpri(1);
    fcntl(fd, F_SETFL, FASYNC|FNDELAY);
    rmask = inputbits;
    emask = wmask = 0;
    agt_create(&sender, SIGIO, &agtmemory);

    for(;;) {
        (void) msg_recv(&sender, &arg, &asz,
          0, 0, INFINITY);
        (void) msg_reply(sender);
        select(32, &rmask, &wmask, &emask,
          (struct timeval *)0);
        if (rmask & inputbits) {
            cnt = read(fd, buf, 256);
            if (cnt != -1 || errno != EWOULDBLOCK ||
              errno != EINTR)
                break;
        }
    }
    buf[cnt] = 0;
    printf("\ngot %s\n", buf);
    fcntl(fd, F_SETFL, 0);
    exit(0);
}
```

2.5. Monitors and Conditions

The monitor-condition variable paradigm is a familiar one to kernel programmers because of the analogue to `sleep()` and `wakeup()` in the UNIX system kernel.

A monitor implements a *critical section.* This is a reentrant region of code in which access is serialized. As a result, shared data accessed by this code is protected against races that can lead to incorrect interpretations of the data. Once a thread is executing within a monitor, other threads block until that monitor is exited. When thread priorities are equal, they are queued first-come-first-served for access to the monitor. This ensures fair, serial access to the protected data.

As an example, a producer and consumer thread may use a monitor to protect access to a buffer of data being produced or consumed (so that the state of the buffer's "fullness" is consistent). When the producer has filled the buffer, it must wait for the consumer to drain the buffer. This sort of synchronization is provided by *condition variables.* When a thread waits on a condition, it atomically gives up the monitor and blocks pending a *notification.* The result of the notification is that the blocked thread will eventually reacquire the monitor in

order to attempt access to the buffer again.

Monitors vs. Interrupt Masking

One goal of lightweight processes is to avoid the use of *sigsetmask's* or other primitives which lock out interrupts to prevent races. By using monitors as a synchronization tool, and by using threads with agents to handle interrupts, the use of interrupt masking can be eliminated, and the risk of dropping interrupts reduced.

Within the LWP library itself, most critical sections are implemented by disabling the scheduler (and *not* by disabling interrupts) for the duration of the critical section. If an interrupt arrives during a critical section, it is processed only to the point of saving the volatile interrupt state. At the end of a critical section, if there are any accumulated events, scheduling decisions are made based upon the agents associated with the events. Interrupts are only masked to ensure that a) the nugget stack is not grown indefinitely by repeated interrupts and b) as a thread is being resumed, to ensure that the new context is loaded atomically. Thus, interrupts are only disabled as a consequence of an interrupt occurring, and never preventively.

Programming with Monitors

Typically, there is some state associated with a condition. When the state acquires a given value, a thread can take some action. Otherwise, it will wait until the state changes. For example, if a buffer is full, a thread writing to the buffer will wait until the state of the buffer indicates that it is no longer full. Another thread reading from the buffer will cooperate by notifying any such waiting thread when the buffer is no longer full. Because the buffer state is accessed by several threads, it is protected by a monitor. Otherwise, a thread could decide to wait for a state change, only to have the state change before the wait can be executed, resulting in deadlock. Therefore, both the waiter and the notifier *must* access the state in a monitor, and the wait primitive `(cv_wait)` must atomically release the monitor. The typical wait code looks like this:

```
mon_enter(m);
...;
while (!state)
    cv_wait(cv);
...;
mon_exit(m);
```

The while loop is there because if there are several threads waiting in the monitor when the condition is broadcast, all of them wake up, but the first thread to gain entry to the monitor may alter the state, invalidating it for the other awakened threads. In our current example, if two producers are awakened because the buffer is no longer full, the first one may fill the buffer again and wait, leaving the second one to run. The second producer must not add to the buffer now, because it is full again.

Some subtle points about thread scheduling priority should be mentioned. Note that threads queue for monitors and conditions based upon thread priority. No context switch necessarily takes place when a monitor is exited. Thus, a monitor that is repeatedly reentered by a high-priority thread can starve other threads

wanting access to the monitor. Care should be taken in assigning priorities to threads using monitors, since a low-priority thread which owns a monitor can still prevent a higher priority thread from accessing that monitor. If a low-priority thread owning a monitor is preempted, it may cause long delays to more important threads needing monitor access.

Monitors and Events

Since events are processed by threads, state manipulated by a thread receiving agent messages can be protected by monitors and condition variables. Thus, after receiving an agent message, a thread may enter a monitor before accessing some global state. Since the LWP library has a large memory for events, no events should be lost if this thread has to block for access to the monitor.

Condition Variables

`cv_broadcast()` awakens *all* threads blocked on a condition. `cv_notify()` awakens only a *single* thread blocked on a condition. `cv_notify()` can result in deadlock states if the awakened thread is not the particular one that should notice a state change and should only be used when it is known that a single other thread is involved. `cv_notify()` is available because it is more efficient to awaken only a single thread. Note that an awakened thread will be queued to reacquire the monitor. When the thread actually resumes, it will own the monitor it released when it waited for the condition with `cv_wait()`.

Because it is both confusing to the programmer and expensive to implement, no provision for a condition to be shared by several monitors is made. Instead, condition variables are bound to a monitor when they are created. It would be possible to let them be bound when the condition is waited upon, but it would allow the very improbable case of having a waiter awaken in a state testing loop, only to find that his condition was reassigned.

`mon_destroy()` will remove any conditions bound to the monitor being removed. If `mon_destroy()` fails because some threads are still waiting on an associated condition, you can use `cv_waiters()` to see which threads are blocked on conditions associated with the monitor, followed by `lwp_destroy()` to terminate the blocked threads. After the offending threads are terminated, `mon_destroy()` should succeed.

Enforcing the Monitor Discipline

Because a thread which forgets to exit a monitor may deadlock the system, it is convenient to use the exception handler mechanism to enforce the enter-exit discipline. The `MONITOR()` macro enforces this discipline by ensuring that `mon_exit()` is called when the procedure that embodies the monitor exits. (It is good form to use a single procedure to contain a monitor, viz:)

```
foo() {
    MONITOR(m);
    ...;
}
```

This method ensures that no matter how the procedure is exited (barring `longjmp()`), the monitor will be exited. That is, if the procedure raises an

exception or returns explicitly or implicitly, the monitor is freed.

Nested Monitors

When a thread blocks on a condition while holding several (*nested*) monitor locks, all of the locks except the current one are held. This ensures that the thread does not need to painfully reacquire all of its locks, with the concomitant possibility of deadlock if not all of the locks remain available. If thread T1 holds monitor M1 and wants to acquire monitor M2, and thread T2 holds monitor M2 and wants to acquire monitor M1, deadlock results. One way to avoid this error is to require that the monitors are always acquired in a certain order.

Reentrant Monitors

When a monitor is used to protect a data structure, it may happen, for information hiding reasons, that two different procedures wish to use the same monitor. It may also happen that one of those procedures wishes to use the facilities provided by the other. If these procedures are accessed by the same thread the monitor calls are *reentrant.* If you anticipate such use, you should program your monitors as

```
if (mon_enter(m) < 0) {
    error("bad monitor");
}
```

However, if you wish to catch reentrant monitor use as an error, you should program monitors as:

```
if (mon_enter(m) != 0) {
    error("reentrant monitor");
}
```

Monitor Program Examples

The following is a simple example of monitor use. As described above, we have a producer and a consumer thread, synchronizing with condition variables. To spice it up a bit, we've added some scheduling to make things more realistic.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

thread_t c1, c2, sched;
mon_t m1;
cv_t notempty, notfull;
int cnt = 0;
int in = 0;
int out = 0;
#define MAXBUF  20
char buf[MAXBUF];
#define MAXPRIO 10

main(argc, argv)
    int argc;
    char **argv;
```

```
{
    int producer(), consumer();
    int sch();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(3000, 3);
    lwp_create(&c1, producer, MINPRIO+1, 0,
      lwp_newstk(), 0);
    lwp_create(&c2, consumer, MINPRIO, 0,
      lwp_newstk(), 0);
    lwp_create(&sched, sch, MAXPRIO, 0, lwp_newstk(), 0);
    mon_create(&m1);
    cv_create(&notempty, m1);
    cv_create(&notfull, m1);
    exit(0);
}

put(c)  /* add a character to the buffer */
    char c;
{
    MONITOR(m1);
    while (cnt == MAXBUF) { /* buffer never > MAXBUF */
        printf("waiting on notfull\n");
        cv_wait(notfull);
    }
    buf[in] = c;
    in = (in + 1) % MAXBUF;
    cnt++;
    cv_broadcast(notempty); /* may be a no-op */
}

get(c)
    char *c;
{
    MONITOR(m1);
    while (cnt == 0) { /* buffer never < 0 chars */
        printf("waiting on notempty\n");
        cv_wait(notempty);
    }
    *c = buf[out];
    out = (out + 1) % MAXBUF;
    cnt--;
    cv_broadcast(notfull);
}

producer() {
    char c;
    int i;
    int j;

    for(j = 0; j < 500; j++) {
        c = "abcdefghijklmnopqrstuvwxyz"[cnt];
        /* produce */
        put(c);
    }
    printf("producer done\n");
```

```
consumer()
{
    char c;
    int i;
    int j;

    for(j = 0; j < 500; j++) {
        get(&c);
        /* consume the character */
    }
    printf("consumer done\n");
}

sch()
{
    int k;
    thread_t x;
    struct timeval wait;

    x = c1;
    wait.tv_sec = 0;
    wait.tv_usec = 100000;

    for(k = 0; k < 100; k++) {
        lwp_sleep(&wait);
        lwp_setpri(x, MINPRIO);
        if (x.thread_id == c1.thread_id)
            x = c2;
        else
            x = c1;
        lwp_setpri(x, MINPRIO+1);
    }
}
```

2.6. Exceptions

The exception primitives can be used to manage synchronous exceptional conditions in a lightweight process. There are no asynchronous exceptions supported by threads because asynchrony can be managed completely with threads and agents, and in a more well-structured fashion. For example, when parsing commands and anticipating an interrupt from the keyboard, you can simply create a thread to parse the command and a thread with an agent to catch the interrupt. When the agent thread catches the interrupt it can simply destroy the parsing thread. This is more elegant than doing a `longjmp()` from a signal handler when an interrupt occurs.

There are several aspects of exceptions. First, you can use *exit_handlers* to be invoked automatically any time a procedure exits. Second, you can provide an exception handler which assumes control anywhere back on the procedure calling chain (*escape exceptions*). Third, you can provide an exception handler which is invoked at the time of an exception and leaves the flow of control alone when it returns (*notification exceptions*). Finally, you can map machine faults (*synchronous traps*) into exceptions. An exception is an event caused by the explicit (or implicit, in the case of synchronous traps) invocation of `exc_raise()`.

When a procedure can exit via a large number of `return` statements or exception raises, it is difficult to monitor the flow of control. Thus, *exit handlers* can be established by `exc_on_exit()` to ensure that a particular action is taken on procedure exit, no matter how the procedure exits. For this reason, no primitive to remove an exit handler is provided, because this provides a way to defeat the whole purpose of exit handlers.

`setjmp()` and `longjmp()` support non-local gotos, but do not give the programmer a disciplined way to invoke them. Pattern-directed handler invocation gives the client an opportunity to establish a set of handlers which are matched by particular patterns. For example, an exception in a memory allocation routine can be raised in such a way that a particular handler (say, a garbage collector) can be explicitly invoked by using a well-known pattern. The `CATCHALL` pattern can be used by a thread either to implement more general sorts of pattern matching (by handling those patterns it wants and discarding those patterns it is not interested in and reraising the exception), or to catch exceptions which must *always* be caught (e.g., a routine which normally allocates some memory permanently and returns should free the memory if an exception occurs).

`exc_notify()` is provided for those exceptions which require an action to be executed on behalf of the exception handler and control to be returned to the raiser of the exception. The handler of a notify exception establishes a function, as well as an argument which can refer to an execution-time environment. By providing a null function, a handler can indicate that only *escape* exceptions (invoked by `exc_raise()`) are to be used.

Exception handling is useful for assisting disciplined use of lightweight process primitives. The `MONITOR()` macro is one example. Another is the `fork()` example discussed in the next section.

Synchronous Traps

Some events are completely synchronous, such as division by zero faults. For such events, it is not logical to allocate a separate thread, since threads are intended to handle asynchronous events. In the lightweight process world, synchronous events appear to be exceptions. Use `agt_trap()` to enable exception mapping for a given event. Note that unhandled exceptions cause termination of the offending thread.

Implementation

One possible way to implement an exception mechanism at the language level would be to use a LWP special context to contain a pointer to the current exception handler for each thread. Using this context, it would be possible to search backwards on the exception chain looking for pattern matches.

Rather than require the client to explicitly pass in a context variable to be used to save and restore exception context, the LWP implementation allocates the context automatically. This is less efficient because by using local variables as contexts, allocation and freeing of the context are free. However, in addition to the more pleasant interface, there are several advantages to the implicit allocation strategy. Because the stack is reset when an exit handler runs, there is no room for local variables to be used by the library code that implements exit handlers (note that the exit handler can make procedure calls of undetermined depth!). This is especially problematic when several exit handlers have been established.

Also, if the system being used can't take interrupts on a separate stack, a fair amount of interrupt masking may be required to protect the stack once it is reset.

Exception handling is really a language issue. However, since synchronous traps may be mapped into exceptions, the LWP library itself must be able to access the exception contexts. Thus, the exception handling facility is part of the LWP library and not a separate language facility. In the future, a more flexible interface to `agt_trap()` may be provided so languages can provide their own style of exception handling.

Example of Exception Handling

In the following example, we use the exception handling mechanisms to facilitate a garbage collector. In the event that a resource is exhausted, the client attempts to correct things by notifying the garbage collector. If the next attempt to obtain the resource fails, the client gives up by raising an exception. As an exercise, pretend that the client had resources that needed to be freed as a result of the fatal exception. Use `CATCHALL` handlers to allow procedures higher up the calling chain to free the resources they allocated.

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>

#define ATTRIBUTE   9
#define FATAL       7
#define MAXPRIO     10

main(argc, argv)
    char **argv;
{
    int task();

    (void)pod_setmaxpri(MAXPRIO);
    lwp_setstkcache(1000, 3);
    (void) lwp_create((thread_t *)0, task, MINPRIO, 0,
      lwp_newstk(), 0);
    exit(0);
}

task()
{
    int garb_collect();

    /* establish garbage collector for ATTRIBUTE-type resources */
    (void) exc_handle(ATTRIBUTE, garb_collect, ATTRIBUTE);

    /* establish handler for unrecoverable errors */
    if (exc_handle(FATAL, 0, 0) == 0)
        someprocedure();
    else
        abort();
}

someprocedure()
{
    char *r;
    char *getresource();
```

```
    r = getresource(ATTRIBUTE);
    /* use resource */
}

char *
getresource(attribute)
    int attribute;
{
    int (*f)();
    char *resource;
    char *obtain();

    resource = obtain(attribute);          /* try to get resource */
    if (resource == 0)  {                  /* couldn't get it */
        (void) exc_notify(attribute);      /* garbage collect */
        resource = obtain(attribute);      /* try again */
        if (resource == 0)                 /* still couldn't get it */
            exc_raise(FATAL);              /* give up */
    }
    return(resource);
}

garb_collect(atr)
    int atr;
{
    /*
     * garbage collect resource of type atr such that
     * obtain might succeed if tried again.
     */
}

char *
obtain(atr)
    int atr;
{
    /*
     * try to allocate resource of type atr
     * return 0 if unable to get the resource.
     */
}
```

2.7. Big Example

This example illustrates many of the LWP features: exit handlers, monitors, condition variables, messages, threads. It is a parallel binary tree fringe comparator. Given two binary trees T1 and T2, they have the same fringe if and only if their leaf nodes are equivalent when read left to right.

Part of the program relies on a `fork()` and `join()` mechanism. The idea is that a thread may wish to start some threads and wait for *n* of them to terminate. (To wait for *one* specific thread to die, use *lwp_join*.) Thus, a program could look like:

```
proc() {
    ...;
    tfork(thread1);
    tfork(thread2);
    tfork(thread3);
    join(2);      /* wait for any 2 tforked threads to die */
    ...;
    join(1);      /* wait for last thread to die */
}
```

To make this work, we have `tfork()` create its thread via an intermediary which uses an exit handler (see `exc_on_exit`(3L)) to ensure that the thread calls `die()` when it terminates. `die()` will keep track of the number of terminated threads. Since a `tfork()`'ed thread may be destroyed by another thread, `lwp_destroy()` should be encapsulated by a procedure that calls `die()` as well. This is an illustration of how the exception handling facility can be used to create new protocols (enforced exit actions, for example).

The program begins by declaring two trees (which don't, in this case, have the same fringe). Then, we create three threads: one thread to evaluate each tree, and one thread to compare leaf values and serve as an information exchanger. The two tree evaluators proceed in parallel, sending a message to the comparator containing the leaf value when a leaf is encountered. When the comparator finds a mismatch, it terminates the tree evaluators. When the main program joins successfully, the two evaluators are dead. It then sends a message to the comparator to find out what the results were.

The tree evaluators are simple: they merely recurse down their subtree, pausing to tell the comparator when a leaf is encountered. The comparator is fairly complex. It first receives a message from *either* of the two tree evaluators (which, after all, are running in parallel. As an exercise, add preemptive round-robin scheduling to this program!). Then, it waits for a message from the *other* tree evaluator (else, it could get another value from the same tree evaluator). If the answers disagree, the comparator terminates the evaluators to prevent further (useless and confusing) messages from being sent. Finally, because the two trees being compared may be structurally quite different, one evaluator may finish while the other remains active. As a result, the comparator could do a `msg_recv()` on a non-existent thread. Therefore, we check this condition by noting if `msg_recv()` fails. Just to show that it's possible, this program lints when linted with the LWP lint library!

```
#include <lwp/lwp.h>
#include <lwp/stackdep.h>
#include <lwp/lwperror.h>
#define NULL 0
thread_t cmp, p1, p2;
thread_t driver;
int tfork();
cv_t cv;
mon_t mon;
```

```
int numdead = 0;
typedef struct tree_t {
    int val;
    struct tree_t *left, *right;
} tree_t;
#define TREENULL ((tree_t *) 0)
#define TRUE 1
#define FALSE 0
#define MAXPRIO 10

tree_t t1[] = {
    {0, &t1[1], &t1[2]},
    {1, &t1[3], &t1[4]},
    {4, TREENULL, TREENULL},
    {1, TREENULL, TREENULL},
    {3, TREENULL, &t1[5]},
    {5, TREENULL, TREENULL},
};

tree_t t2[] = {
    {0, &t2[1], &t2[2]},
    {1, TREENULL, TREENULL},
    {2, &t2[3], &t2[4]},
    {3, TREENULL, TREENULL},
    {4, TREENULL, TREENULL},
};

main()
{
    int compare(), parsetree();
    int answer;

    if (pod_setmaxpri(MAXPRIO) == -1)
        lwp_perror("setmaxpri");
    (void)lwp_setstkcache(10000, 5);
    (void)lwp_self(&driver);
    tfork(&cmp, compare, 0);
    tfork(&p1, parsetree, (int)t1);
    tfork(&p2, parsetree, (int)t2);
    join(2);
    (void)msg_send(cmp, (caddr_t)0, 0,
      (caddr_t)&answer, sizeof (answer));
    if (answer)
        (void) printf("same fringe\n");
    else
        (void) printf("not same fringe\n");
    exit(0);
}

compare()
{
    int val1;
    thread_t next;
    thread_t sender;
    int samefringe = TRUE;
    int *resbuf;
```

```
    int ressize;
    int *argbuf;
    int argsize;
    int err;

    for(;;) {
        err = MSG_RECVALL(&sender, (caddr_t *)&argbuf,
          &argsize, (caddr_t *)&resbuf,
          &ressize, INFINITY);
        if (err < 0)
            lwp_perror("MSG_RECVALL");
        if (SAMETHREAD(sender, driver)) {
            *resbuf = samefringe;
            (void) msg_reply(driver);
            return;
        }
        val1 = *argbuf;
        next = (SAMETHREAD(sender, p1) ? p2 : p1);
        (void) msg_reply(sender);
        err = msg_recv(&next, (caddr_t *)&argbuf,
          &argsize, (caddr_t *)&resbuf,
          &ressize, INFINITY);
        if (err < 0) { /* he died */
            samefringe = FALSE;
            destroy(sender);
        } else {
            samefringe = (*argbuf == val1);
            if (!samefringe) {
                destroy(p1);
                destroy(p2);
            } else
                (void)msg_reply(next);
        }
    }
}

parsetree(t)
    tree_t *t;
{
    if (t == TREENULL)
        return;
    if ((t->left == TREENULL) && (t->right == TREENULL)) {
        /* leaf */
        (void)msg_send(cmp, (caddr_t)&t->val,
          sizeof (int), (caddr_t)0, 0);
    } else {
        parsetree(t->left);
        parsetree(t->right);
    }
}

tfork(new, adr, arg)
    thread_t *new;
    int (*adr)();
    int arg;
```

```
{
    extern void prochelp();
    static int init = 0;

    if (init == 0) {
        init = 1;
        (void)mon_create(&mon);
        (void)cv_create(&cv, mon);
    }
    (void)lwp_create(new, prochelp, MINPRIO, 0,
      lwp_newstk(), 2, adr, arg);
}

void
prochelp(proc, arg)
    int (*proc)();
{
    extern void die();

    (void)exc_on_exit(die, (caddr_t)0);
    proc(arg);
}

void
die()
{
    MONITOR(mon);
    numdead++;
    (void)cv_notify(cv);
}

join(cnt)
{
    MONITOR(mon);
    while (numdead < cnt)
        (void)cv_wait(cv);
    numdead -= cnt;
}

/* use this instead of lwp_destroy with tfork and join */
destroy(pid)
    thread_t pid;
{
    die();
    (void)lwp_destroy(pid);
}
```

3

System V Interprocess Communication Facilities

3.1. IPC Facilities in the SunOS Operating System

Interprocess Communication involves sharing data between processes and, when necessary, coordinating access to the shared data. Release 4.1 of the SunOS operating system (referred to hereafter as "Release 4.1," or "4.1") provides a number of facilities and mechanisms by which processes can communicate.

File I/O and Pipes

In the simplest case, processes can communicate by writing to and reading information from files. Alternatively, a process may provide data for direct consumption by another concurrent process using a *pipe*. Pipes employ the basic byte-stream model used for file I/O.

State Files and File Locking

A process may deposit context in a state file for use by a later invocation. Processes that make use of state files can prevent multiple concurrent access (and race conditions on writes), by using lock files to simulate semaphores. Before attempting to open a state file for write access, a program can test for the existence of a lock file, to determine whether the desired file or device is available. A simple way to create a lock file is to use the `open`(2) system call with the `O_CREAT` and `O_EXCL`, flags. When called in this way, `open()` creates the lock file only if it does not already exist. If multiple processes both attempt to get a lock at about the same time, only the first will succeed. The other processes may be instructed to block (suspend execution) until such time as the lock file is removed, or to exit with an appropriate error message.

Lock files are most useful when the lock is to persist through a reboot of the system. A case in point is the permissions file used by SCCS.

Additionally, the system provides library routines such as `flock`(3) and `lockf`(3) for advisory or mandatory file locking. Locks placed with `flock()` are only visible to processes running on the local processor. Locks placed with `lockf()` are visible to any process running on any processor with access to the file. `lockf()` also provides record locking for fine-grained control over updates to specific regions (strings of contiguous bytes) within a file.

Named Pipes

Another facility that makes use of the file system for IPC is the System V named pipe mechanism. A named pipe (also referred to as a FIFO) has an entry in the file system, but otherwise behaves like an ordinary pipe. It allows one process to provide output directly to another process through ordinary reads and writes to the named device. Unlike ordinary pipes, when the processes terminate, the named pipe remains available for use by other processes. (Refer to `mknod`(8) for

more information.)

Named pipes suffer from all the limitations of regular pipes. For instance, the sender is unknown to the process reading the pipe. Unfortunately, this allows multiple processes to interleave output. Input from a named pipe should therefore be used with caution.

Networking Facilities

Release 4.1 supports two important facilities for networking and IPC in general. They are: TLI (from System V) and sockets (from BSD). These facilities, which both support the file I/O (byte stream) model, can be used for IPC on the local host. They are often preferred when a service has both local and network clients. For more information about networking and general IPC facilities, refer to *Network Programming*.

3.2. System V IPC Facilities in Release 4.1

Release 4.1 provides the following System V facilities for memory-based IPC on a local system:

- Messages
- Semaphores
- Shared Memory

Relying on the native virtual memory manager, in conjunction with the `mmap`(2) system call, often provides better performance for shared access to read-only segments in memory.

These facilities allow local processes to share and process messages, to share access to memory segments in a manner that is compatible with existing System V applications, and to coordinate access to shared objects.

If the process that creates an IPC facility dies, the facility does *not* expire along with it; an IPC facility must be removed explicitly. A shared memory segment remains active, even after it has been flagged for removal, as long as it is attached anywhere in the address space of any process. Only after the last attachment is released, is the (detached) segment freed.

Configuring System V IPC Facilities

In order to use these facilities, they must be configured into your kernel. The relevant configuration options are:

`IPCMESSAGE` for the System V Messages facility.

`IPCSEMAPHORE` for the System V Semaphore facility.

`IPCSHMEM` for the System V Shared Memory facility.

For details on how to configure a kernel, refer to *System and Network Administration*.

System V IPC Permissions

Permissions for a System V IPC facility can be extended to users other than the one for which the facility was created. The creating process identifies the default owner. Unlike files, however, the creator can assign ownership of the facility to another user; it can also revoke an ownership assignment. The current owner process, in turn, can grant read or write access to still other users.

The definition for the IPC permissions data structure `ipc_perm`, is given in `<sys/ipc.h>`, as shown below.

Figure 3-1 *IPC Permissions Data Structure*

```
struct ipc_perm
{
    ushort  uid;    /* owner's user id */
    ushort  gid;    /* owner's group id */
    ushort  cuid;   /* creator's user id */
    ushort  cgid;   /* creator's group id */
    ushort  mode;   /* access modes */
    ushort  seq;    /* slot usage sequence number */
    key_t   key;    /* key */
};
```

This structure is common to all System V IPC facilities. Permissions for an IPC facility are initialized by the creating process, and can be modified by any process that has permission to perform control operations on that facility. Permissions are specified as octal values in the flags argument of the appropriate IPC creation or control system call:

Figure 3-2 *IPC Permission Modes*

Access Permissions	*Octal Value*
Write by Owner	0200
Read by Owner	0400
R/W by Owner	0600
Write by Group	0020
Read by Group	0040
R/W by Group	0060
Write by Others	0002
Read by Others	0004
R/W by Others	0006

For instance, if read access by the owner, and read/write by others is desired, the permissions value would be `0406`.

IPC System Calls, Key Arguments, and Creation Flags

Multiple processes requesting access to a common IPC facility must have a means for determining the identity of the desired facility. To that end, system calls that initialize or provide access to an IPC facility make use of a *key* argument (of type `key_t`). This key is a value that is either known to all the programs, or preferably, one that can be derived from a common seed at run time. The typical method for deriving a key is to use `ftok`(3) to convert a convenient filename to a suitable value. The value derived is virtually unique within the system. It can be used by all programs (processes) that attempt to obtain access to the facility.

System calls that initialize or get access to a System V IPC facility return an ID number (of type `int`). This ID is used by IPC system calls that perform read, write and control operations, once the facility's ID has been acquired.

If the key argument is specified as `IPC_PRIVATE` (defined to be zero), the call initializes a new instance of an IPC facility that is private to the creating process.

When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the system call attempts to create the facility if it does not exist already.

When called with both `IPC_CREAT` and `IPC_EXCL` flags, the system call fails if the facility already exists. This can be useful when more than one process may attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds.

If neither of these flags is given, and the facility already exists, the system calls to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail.

These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example:

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

initializes a new message queue, but only if the queue does not exist already. The first argument evaluates to a key based on the string; the second, the combined permissions and control flags.

System V IPC Configuration Options

A number of system configuration options* for data structures used by System V IPC facilities can be adjusted in the system configuration file. Some of these options set limits on the amount of resources avaliable to an IPC facility. Those that affect specific system calls are discussed in the descriptions of those system calls. For more information about System V IPC configuration options, you may wish to refer to *System and Network Administration.*

3.3. Messages

The System V messaging facility provides processes with a means to send and receive messages, and to queue messages for processing in an arbitrary order. Unlike the typical file byte-stream model of data flow (in sockets and TLI), System V messages each have an explicit length. More importantly, messages can be assigned a specific type. Among other uses, this allows a server process to direct message traffic between multiple clients on its queue (by using the PID of the client process as the message type). For operations involving single-message transactions, a server can balance the load between multiple server processes that have access to the queue.

Before a process can send or receive a message, the queue must be initialized by making an `msgget(2)` system call. The owner or creator of a queue can change its ownership or permissions using `msgctl(2)`. In addition, any process with permission to do so can use `msgctl()` to perform control operations.

Refer to `config(8)` and *Installing SunOS 4.1* for information on how to configure a SunOS operating system kernel.

Operations to send and receive messages are performed respectively by the `msgsnd()` and `msgrcv()` system calls (see `msgop`(2)). When a message is sent, its text is copied to the message queue.

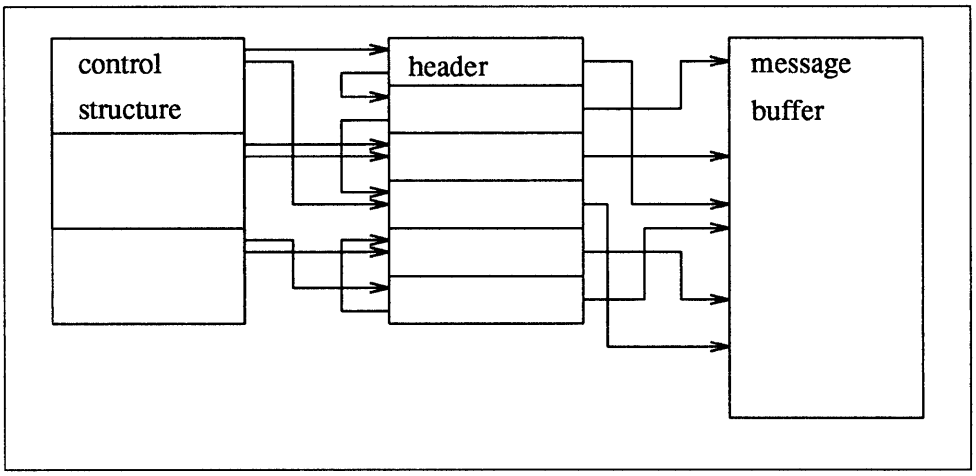
`msgsnd()` and `msgrcv()` can be performed as either blocking, or non-blocking operations. A blocked message operation remains suspended until one of three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Structure of a Message Queue

A message queue is composed of a control structure with a unique ID, a linked list of message headers, and a buffer in which to store the text of the message(s). The identifier for the queue is referred to as the `msqid`.

Figure 3-3 *Structure of a Message Queue*



The control structure for the message queue contains the following information:

- A permissions structure.
- A pointer to the first message on the queue.
- A pointer to the last message on the queue.
- The current number of bytes in the queue.
- The number of messages in the queue.
- The maximum number of bytes allowed in the queue.
- The process ID (PID) of last message sender.
- The PID of last message receiver.
- The time the last message was sent.
- The time the last message was received.

- The time of the last change to the structure.

Each message header contains the following information:

- A pointer to the next message on the queue.
- The message type.
- The message text size.
- The message text address.

The message queue control structure is defined in the header file `<sys/msg.h>`:

Figure 3-4 *Message Queue Control Structure*

```
struct msqid_ds
{
        struct ipc_perm  msg_perm;    /* access permission struct */
        struct msg       *msg_first;  /* ptr to first message on q */
        struct msg       *msg_last;   /* ptr to last message on q */
        ushort           msg_cbytes;  /* current # bytes on q */
        ushort           msg_qnum;    /* # of messages on q */
        ushort           msg_qbytes;  /* max # of bytes on q */
        ushort           msg_lspid;   /* pid of last msgsnd */
        ushort           msg_lrpid;   /* pid of last msgrcv */
        time_t           msg_stime;   /* last msgsnd time */
        time_t           msg_rtime;   /* last msgrcv time */
        time_t           msg_ctime;   /* last change time */
};
```

Likewise, the definition for the message-header data structure is given as:

Figure 3-5 *Message Header Structure*

```
struct msg
{
    struct msg  *msg_next;  /* ptr to next message on q */
    long        msg_type;   /* message type */
    short       msg_ts;     /* message text size */
    short       msg_spot;   /* message text map address */
};
```

Initializing a Message Queue with `msgget()`

The `msgget()` system call is used to initialize a new message queue. It can also be used to return the message queue ID (`msqid`) of the existing queue that corresponds to the key argument. When the call fails, it returns –1, and sets the external variable `errno` to the appropriate error code. `msgget()` has the synopsis shown below.

Figure 3-6 *Synopsis of* `msgget()`

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int msgget(key, msgflg)
key_t    key;
int      msgflg;
```

The value passed as the `msgflg` argument must be an octal integer, which incorporates settings for the queue's permissions and control flags, as described under *System V IPC Permissions*, above.

The `MSGMNI` kernel configuration option determines the maximum number of unique message queues that the kernel will support. `msgget()` fails when this limit is exceeded.

The following example is a simple exerciser to illustrate the `msgget()` system call. The program begins by prompting for a key, an octal permissions code, and finally, for your choice of control flags. It allows all possible combinations. If `msgget()` fails, the program indicates that there was an error, and displays the value of `errno`. Otherwise, it displays the message queue ID that the call returned.

Figure 3-7 *Sample Program to Illustrate* `msgget()`

```
/*
** msgget.c:  Illustrate the msgget() system call.
**
**   This is a simple exerciser of the msgget() system call.
** It prompts for the arguments, makes the call, and reports the
** results.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

extern void    exit();
extern void    perror();

main()
{
     key_tkey; /* key to be passed to msgget() */
     int  msgflg,    /* msgflg to be passed to msgget() */
          msqid;     /* return value from msgget() */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
     (void) fprintf(stderr, "Enter desired key: ");
```

```
        (void) scanf("%li", &key);

        (void) fprintf(stderr, "\nExpected flags for msgflg argument are:\n");
        (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
        (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
        (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
        (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
        (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
        (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
        (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
        (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
        (void) fprintf(stderr, "Enter desired msgflg value: ");
        (void) scanf("%i", &msgflg);

        (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx, %#o)\n",
             key, msgflg);
        if ((msqid = msgget(key, msgflg)) == -1)
        {
             perror("msgget: msgget failed");
             exit(1);
        } else {
             (void) fprintf(stderr,
                  "msgget: msgget succeeded: msqid = %d\n", msqid);
             exit(0);
        }
        /* NOTREACHED */
   }
```

Controlling Message Queues with `msgctl()`

The `msgctl()` system call is used to alter the permissions and other characteristics of a message queue. Its synopsis is as follows:

Figure 3-8 *Synopsis of* `msgctl()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(msqid, cmd, buf)
int     msqid, cmd;
struct  msqid_ds *buf;
```

Upon successful completion, the call returns zero. It returns –1 on failure, and sets `errno` appropriately.

The `msqid` argument must be the ID of an existing message queue. The `cmd` argument is one of the following:

`IPC_STAT` Place information about the status of the queue in the the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

`IPC_SET` Set the owner's user and group ID, the permissions, and the size (number of bytes) of the message queue. A process must have the effective user ID of the owner, creator or the super-user for this call to succeed.

`IPC_RMID` Remove the message queue specified by the `msqid` argument.

The following sample program illustrates the `msgctl`(2) system call with all its various flags.

Figure 3-9 *Sample Program to Illustrate* `msgctl()`

```
/*
** msgctl.c:  Illustrate the msgctl() system call.
**
**   This is a simple exerciser of the msgctl() system call.  It allows
** you to perform one control operation on one message queue.  It
** gives up immediately if any control operation fails, so be careful not
** to set permissions to preclude read permission; you won't be able to reset
** the permissions with this code if you do.
*/

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>
#include  <time.h>

static void    do_msgctl();
extern void    exit();
extern void    perror();
static char    warning_message[] = "If you remove read permission for\
 yourself, this program will fail frequently!";

main()
{
     struct msqid_dsbuf; /* queue descriptor buffer for IPC_STAT
                         and IPC_SET commands */
     int       cmd, /* command to be given to msgctl() */
               msqid;    /* queue ID to be given to msgctl() */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     /* Get the msqid and cmd arguments for the msgctl() call. */
     (void) fprintf(stderr,
          "Please enter arguments for msgctl() as requested.");
     (void) fprintf(stderr, "\nEnter the desired msqid: ");
     (void) scanf("%i", &msqid);

     (void) fprintf(stderr, "Valid msgctl commands are:\n");
     (void) fprintf(stderr, "\tIPC_RMID =  %d\n", IPC_RMID);
     (void) fprintf(stderr, "\tIPC_SET  =  %d\n", IPC_SET);
     (void) fprintf(stderr, "\tIPC_STAT =  %d\n", IPC_STAT);
     (void) fprintf(stderr, "\nEnter the value for the desired command: ");
     (void) scanf("%i", &cmd);

     switch (cmd) {
     case IPC_SET:
          /* Modify settings in the message queue control structure. */
          (void) fprintf(stderr, "Before IPC_SET, get current values:");
          /* fall through to IPC_STAT processing */

     case IPC_STAT:
          /*
          ** Get a copy of the current message queue control structure
          ** and show it to the user.
```

```
            */
            do_msgctl(msqid, IPC_STAT, &buf);
            (void) fprintf(stderr,
                 "msg_perm.uid = %d\n", buf.msg_perm.uid);
            (void) fprintf(stderr,
                 "msg_perm.gid = %d\n", buf.msg_perm.gid);
            (void) fprintf(stderr,
                 "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
            (void) fprintf(stderr,
                 "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
            (void) fprintf(stderr, "msg_perm.mode = %#o, ",
                 buf.msg_perm.mode);
            (void) fprintf(stderr, "access permissions = %#o\n",
                 buf.msg_perm.mode & 0777);
            (void) fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
            (void) fprintf(stderr, "msg_qbytes = %d\n", buf.msg_qbytes);
            (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
            (void) fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
            (void) fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
            (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
                 ctime(&buf.msg_stime) : "Not Set\n");
            (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
                 ctime(&buf.msg_rtime) : "Not Set\n");
            (void) fprintf(stderr, "msg_ctime = %s", ctime(&buf.msg_ctime));
            if (cmd == IPC_STAT)
                 break;

            /*
            **   Now continue with IPC_SET.
            */
            (void) fprintf(stderr, "Enter desired msg_perm.uid: ");
            (void) scanf ("%hi", &buf.msg_perm.uid);

            (void) fprintf(stderr, "Enter desired msg_perm.gid: ");
            (void) scanf("%hi", &buf.msg_perm.gid);

            (void) fprintf(stderr, "%s\n", warning_message);
            (void) fprintf(stderr, "Enter desired msg_perm.mode: ");
            (void) scanf("%hi", &buf.msg_perm.mode);

            (void) fprintf(stderr, "Enter desired msg_qbytes: ");
            (void) scanf("%hi", &buf.msg_qbytes);

            do_msgctl(msqid, IPC_SET, &buf);
            break;

       case IPC_RMID:
       default:
            /* Remove the message queue or try an unknown command. */
            do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
            break;
       }
       exit(0);
       /* NOTREACHED */
  }

  /*
  **   Print indication of arguments being passed to msgctl(), call msgctl(),
  ** and report the results.
  **   If msgctl() fails, do not return; this example doesn't deal with
  ** errors, it just reports them.
  */
  static void
  do_msgctl(msqid, cmd, buf)
  struct msqid_ds*buf;
  int       cmd,
            msqid;
```

```
{
        register int    rtrn;/* hold area for return value from msgctl() */

        (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
                msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
        rtrn = msgctl(msqid, cmd, buf);
        if (rtrn == -1) {
                perror("msgctl: msgctl failed");
                exit(1);
                /* NOTREACHED */
        } else {
                (void) fprintf(stderr, "msgctl: msgctl returned %d\n", rtrn);
        }
}
```

Sending and Receiving Messages with `msgsnd()` and `msgrcv()`

`msgsnd`(2) and `msgrcv`(2) are used to send and receive messages, respectively. Their synopses are as follows:

Figure 3-10 *Synopses of* `msgsnd()` *and* `msgrcv()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(msqid, msgp, msgsz, msgflg)
int     msqid;
struct  msgbuf *msgp;
int     msgsz, msgflg;

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Upon successful completion, these system calls each return zero; when unsuccessful, they return –1, and set the external variable `errno` to the appropriate error code.

The `msqid` argument must be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The `msgsz` argument specifies the length of the message (in bytes).

Various control flags can be passed in the `msgflg` argument. Flags can be combined within the argument using logical OR operator. If `IPC_NOWAIT` is set, a send or receive operation that cannot complete will fail. For instance, a non-blocking `msgrcv()` operation will fail if there is no message to receive. If `MSG_NOERROR` is set, then a message longer than the size specified by `msgsz` is truncated to that size. Note that the trailing portion of the truncated message is lost. Without the `MSG_NOERROR` flag, attempting to receive a message that is longer than expected results in failure.

The `msgtyp` argument to `msgrcv()` is used to indicate the type of message to receive. If this argument is equal to zero, the call receives the first message on the queue. If it is greater than zero, the call receives the first message of the indicated type.

If `msgtyp` is less than zero, the call receives the first extant message on the queue with lowest type value, up to and including the absolute value of the argument. For instance, if `msgtyp` has a value of –3, the call retrieves the first message of type 1, if any, or the first message of type 2, if any, or the first message of type 3. It would not receive a message of type 4. This allows you to prioritize message processing according to type.

The following sample program illustrates `msgsnd()` and `msgrcv()`.

Figure 3-11 *Sample Program to Illustrate* `msgsnd()` *and* `msgrcv()`

```
/*
** msgop.c: Illustrate the msgsnd() and msgrcv() system calls.
**
**   This is a simple exerciser of the message send and receive
** routines.  It allows the user to attempt to send and receive as many
** messages as desired to or from one message queue.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

static intask();
extern void    exit();
extern char    *malloc();
extern void    perror();

char first_on_queue[] = "-> first message on queue",
     full_buf[] = "Message buffer overflow.  Extra message text discarded.";

main()
{
     register int   c;         /* message text input */
     int       choice;         /* user's selected operation code */
     register int   i;         /* loop control for mtext */
     int       msgflg;         /* message flags for the operation */
     struct msgbuf  *msgp;          /* pointer to the message buffer */
     int       msgsz;          /* message size */
     long      msgtyp;         /* desired message type */
     int       msqid,          /* message queue ID to be used */
               maxmsgsz, /* size of allocated message buffer */
               rtrn;     /* return value from msgrcv or msgsnd */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     /* Get the message queue ID and set up the message buffer. */
     (void) fprintf(stderr, "Enter desired msqid: ");
     (void) scanf("%i", &msqid);

     /*
     **   Note that <sys/msg.h> includes a definition of struct msgbuf
     ** with the mtext field defined as:
```

```
**        char mtext[1];
** therefore, this definition is only a template, not a directly
** useable structure definition, unless you only want to send
** and receive messages of 0 or 1 byte.
**   To handle this, we malloc an area big enough to contain the
** template - the size of the mtext template field + the size of
** the mtext field we want.  Then we can use the pointer returned
** by malloc as a struct msgbuf with an mtext field of the size
** we want.
**   Note also that sizeof msgp->mtext is valid even though msgp
** isn't pointing to anything yet.  Sizeof doesn't dereference msgp,
** it just uses its type to figure out what we are asking about.
*/
(void) fprintf(stderr, "Enter the message buffer size you want: ");
(void) scanf("%i", &maxmsgsz);
if (maxmsgsz < 0) {
     (void) fprintf(stderr, "msgop: %s\n",
          "The message buffer size must be >= 0.");
     exit(1);
     /* NOTREACHED */
}
msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf) -
     sizeof msgp->mtext + maxmsgsz));
if (msgp == NULL) {
     (void) fprintf(stderr, "msgop: %s %d byte messages\n",
          "could not allocate message buffer for", maxmsgsz);
     exit(1);
     /* NOTREACHED */
}

/* Loop through message operations until the user is ready to quit. */
while (choice = ask()) {
     switch (choice) {
     case 1: /* msgsnd() requested:  Get the arguments, make the
                call, and report the results.  */
          (void) fprintf(stderr, "Valid msgsnd message %s\n",
               "types are positive integers.");
          (void) fprintf(stderr, "Enter desired msgp->mtype: ");
          (void) scanf("%li", &msgp->mtype);

          if (maxmsgsz) {
               /* Since we've been using scanf, we need the
                  following loop to throw away the rest of
                  the input on the line after the entered
                  mtype before we start reading the mtext. */
               while ((c = getchar()) != '\n' && c != EOF)
                    ;

               (void) fprintf(stderr, "Enter a %s:\n",
                    "one line message");
               for (i = 0; ((c = getchar()) != '\n'); i++) {
                    if (i >= maxmsgsz) {
                         (void) fprintf(stderr,
                              "\n%s\n", full_buf);
                         while ((c = getchar()) != '\n')
                              ;
                         break;
                    }
                    msgp->mtext[i] = c;
               }
               msgsz = i;
          } else
               msgsz = 0;

          (void) fprintf(stderr,
               "\nMeaningful msgsnd flag is:\n");
```

```
                (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
                     IPC_NOWAIT);
                (void) fprintf(stderr, "Enter desired msgflg: ");
                (void) scanf("%i", &msgflg);

                (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
                     "msgop: Calling msgsnd", msqid, msgsz, msgflg);
                (void) fprintf(stderr, "msgp->mtype = %ld\n",
                     msgp->mtype);
                (void) fprintf(stderr, "msgp->mtext = \"");
                for (i = 0; i < msgsz; i++)
                     (void) fputc(msgp->mtext[i], stderr);
                (void) fprintf(stderr, "\"\n");

                rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
                if (rtrn == -1)
                     perror("msgop: msgsnd failed");
                else
                     (void) fprintf(stderr,
                          "msgop: msgsnd returned %d\n", rtrn);
                break;

           case 2: /* msgrcv() requested:  Get the arguments, make the
                      call, and report the results.  */
                for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
                     (void) scanf("%i", &msgsz))
                     (void) fprintf(stderr,
                          "%s (0 <= msgsz <= %d): ",
                          "Enter desired msgsz", maxmsgsz);

                (void) fprintf(stderr, "msgtyp meanings:\n");
                (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
                (void) fprintf(stderr, "\t>0 %s of given type\n",
                     first_on_queue);
                (void) fprintf(stderr,
                     "\t<0 %s with type <= |msgtyp|\n",
                     first_on_queue);
                (void) fprintf(stderr, "Enter desired msgtyp: ");
                (void) scanf("%li", &msgtyp);

                (void) fprintf(stderr,
                     "Meaningful msgrcv flags are:\n");
                (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
                     MSG_NOERROR);
                (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
                     IPC_NOWAIT);
                (void) fprintf(stderr, "Enter desired msgflg: ");
                (void) scanf("%i", &msgflg);

                (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
                     "msgop: Calling msgrcv",
                     msqid, msgsz, msgtyp, msgflg);

                rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

                if (rtrn == -1)
                     perror("msgop: msgrcv failed");
                else {
                     (void) fprintf(stderr, "msgop: %s %d\n",
                          "msgrcv returned", rtrn);
                     (void) fprintf(stderr, "msgp->mtype = %ld\n",
                          msgp->mtype);
                     (void) fprintf(stderr, "msgp->mtext is: \"");
                     for (i = 0; i < rtrn; i++)
                          (void) fputc(msgp->mtext[i], stderr);
                     (void) fprintf(stderr, "\"\n");
                }
                break;
```

```
            default:
                (void) fprintf(stderr, "msgop: operation unknown\n");
                break;
            }
        }
        exit(0);
        /* NOTREACHED */
}

/*
**  Ask user what to do next.  Return the user's choice code.
**  Don't return until the user selects a valid choice.
*/
static
ask()
{
        int  response; /* User's response. */

        do {
            (void) fprintf(stderr, "Your options are:\n");
            (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
            (void) fprintf(stderr, "\tmsgsnd =\t1\n");
            (void) fprintf(stderr, "\tmsgrcv =\t2\n");
            (void) fprintf(stderr, "Enter your choice: ");

            /* Preset response so "^D" will be interpreted as exit. */
            response = 0;
            (void) scanf("%i", &response);
        } while (response < 0 || response > 2);

        return(response);
}
```

3.4. Semaphores

Semaphores provide a mechanism by which processes can query or alter status information. They are often used to monitor and control the availability of system resources, such as System V shared memory segments. Semaphores may be operated on as individual units, or as elements in a set. A semaphore set consists of a control structure and an array of individual semaphores. By default, a set of semaphores may contain up to 25 elements; this limit can be altered using the `SEMMSL` system configuration option.

Before a process can use a semaphore, the semaphore set must be initialized using `semget`(2). The semaphore's owner or creator can change its ownership or permissions using `semctl`(2). In addition, any process with permission to do so can use `semctl()` to perform control operations. Semaphore operations are performed by the `semop`(2) system call. This call accepts a pointer to an array of semaphore operation structures; each structure in the operations array contains information about an operation to perform on a semaphore. The operations array is described in detail under *Semaphore Operations*, below.

Any process with read permission can test to see whether a semaphore has a zero value, by supplying a 0 in the `sem_op` field of the operation structure. Operations to increment or decrement a semaphore require alter permission (that is, write permission).

If an attempt to perform any of the requested operations should fail, none of the semaphores are altered. The process will block (unless the `IPC_NOWAIT` flag is set), and will remain blocked until one of the following occurs:

- the semaphore operations can all complete, in which case the call succeeds
- the process receives a signal, or
- the semaphore set is removed.

If a nonblocking semaphore operation fails, the call returns –1 and sets `errno` appropriately.

Only one process can update a semaphore set at any given time. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, the updates are made atomically. That is, no updates are committed until all operations in the array can complete successfully.

Once a process performs an operation on a semaphore, the system does not keep track of whether or not that operation has been undone. If a process with exclusive use of a semaphore terminates abnormally and neglects to undo the operation or free the semaphore, the semaphore will remain locked in memory. To prevent this, `semop()` accepts the `SEM_UNDO` control flag. When this flag is in effect, `semop()` allocates an *undo* structure for each semaphore operation. That structure contains the operation needed to return the semaphore to its previous state. When the process dies, the system applies the operations in the undo structures. That way an aborted process need not leave a semaphore set in an inconsistent state.

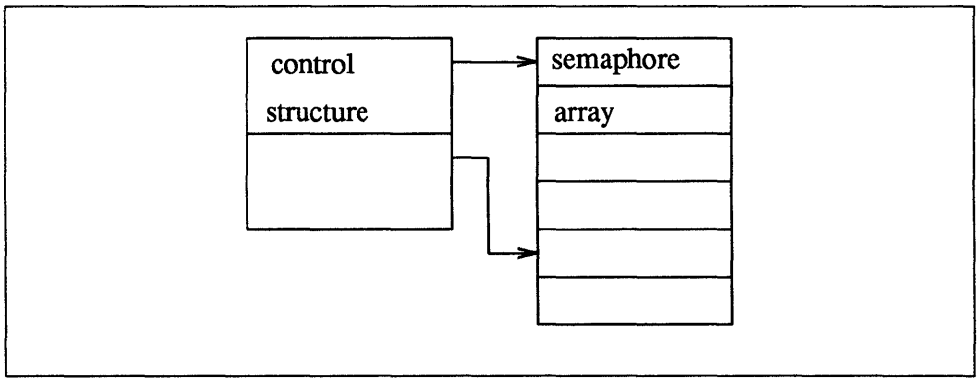
If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this in order to restore the resource to a consistent state.

When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that would perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This insures that, unless the process is aborted, the values applied to the undo structure will eventually cancel out to zero. When the undo structure reaches zero, it is removed. Using `SEM_UNDO` inconsistently can lead to undue resource consumption, since undo structures which are allocated may not be freed (until the system is rebooted).

Structure of a Semaphore Set

A semaphore set is composed of a control structure with a unique ID, along with an array of semaphores. The identifier for the semaphore or array is referred to as the `semid`.

Figure 3-12 *Structure of a Semaphore*



The control structure for the semaphore contains the following information:

- The permissions structure.
- A pointer to first semaphore in the array.
- The number of semaphores in the array.
- The time of the last operation on any semaphore the array.
- The time of the last update to any semaphore in the array.

Each semaphore structure in the array, contains the following information:

- The semaphore value.
- The PID of the process performing the last successful operation.
- The number of processes waiting for the semaphore to increase.
- The number of processes waiting for the semaphore to reach zero.

The control structure is defined in the header file: `<sys/sem.h>`:

```
struct semid_ds
{
    struct ipc_perm sem_perm;   /* permission struct */
    struct sem      *sem_base;  /* ptr to first semaphore in set */
    ushort          sem_nsems;  /* # of semaphores in set */
    time_t          sem_otime;  /* last semop time */
    time_t          sem_ctime;  /* last change time */
};
```

The `sem_perm` member of this structure uses `ipc_perm` (defined in `<sys/ipc.h>`) as a template.

The semaphore structure is defined as:

```
struct sem
{
    ushort  semval;     /* semaphore text map address */
    short   sempid;     /* pid of last operation */
    ushort  semncnt;    /* # awaiting semval > cval */
    ushort  semzcnt;    /* # awaiting semval = 0 */
};
```

in that header file as well.

Initializing a Semaphore Set with `semget()`

The `semget()` system call is used to initialize or gain access to a semaphore. When the call succeeds, it returns the semaphore ID (`semid`). When the call fails, it returns –1, and sets the external variable `errno` to the appropriate error code. `semget()` has the following synopsis:

Figure 3-13 *Synopsis of* `semget()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key, nsems, semflg)
key_t   key;
int     nsems, semflg;
```

As noted above, the key argument is a value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails if `nsems` is greater than the number of elements in an existing array; when the correct count not known, supplying 0 for this argument assures that it will succeed. The `semflg` argument is used to specify the initial access permissions and creation control flags.

The `SEMMNI` system configuration option determines the maximum number of semaphore arrays allowed. The `SEMMNS` option determines the maximum possible number of individual semaphores in across all semaphore sets. `semget()` fails when one of these limits would be exceeded. Due to fragmentation between semaphore sets, you may not be able to allocate all available semaphores.

The following program illustrates the `semget()` system call. It begins by prompting for a hexadecimal key, an octal permissions code, and control command combinations selected from a menu. All possible combinations are allowed.

It then requests the number of semaphores in the array, and issues the system call to initialize the array. If the call succeeds, the program displays the semaphore ID returned. Otherwise, it displays an error message.

Figure 3-14 *Sample Program to Illustrate* `semget()`

```
/*
** semget.c:  Illustrate the semget() system call.
**
**   This is a simple exerciser of the semget() system call.
** It prompts for the arguments, makes the call, and reports the
** results.
*/

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

extern void    exit();
extern void    perror();

main()
{
     key_tkey; /* key to be passed to semget() */
     int  semflg;   /* semflg to be passed to semget() */
     int  nsems;    /* nsems to be passed to semget() */
     int  semid;    /* return value from semget() */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
     (void) fprintf(stderr, "Enter desired key: ");
     (void) scanf("%li", &key);

     (void) fprintf(stderr, "Enter desired nsems value: ");
     (void) scanf("%i", &nsems);

     (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
     (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
     (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
     (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
     (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
     (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
     (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
     (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
     (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
     (void) fprintf(stderr, "Enter desired semflg value: ");
     (void) scanf("%i", &semflg);

     (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %d, %#o)\n",
          key, nsems, semflg);

     if ((semid = semget(key, nsems, semflg)) == -1) {
          perror("semget: semget failed");
          exit(1);
     } else {
          (void) fprintf(stderr, "semget: semget succeeded: semid = %d\n",
               semid);
          exit(0);
     }
     /*NOTREACHED*/
}
```

Controlling Semaphores with `semctl()`

The `semctl()` system call allows a process to alter permissions and other characteristic of a semaphore set. Its synopsis is as follows:

Figure 3-15 *Synopsis of* `semctl()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(semid, semnum, cmd, arg)
int     semid, cmd;
int     semnum;
union   semun
{
        int val;
        struct semid_ds *buf;
        ushort * array;
} arg;
```

`semid` is a valid sempahore ID. `semnum` is used to select a semaphore within an array by its index. The `cmd` argument is one of the following control flags. What you supply for `arg` depends upon the control flag given in `cmd`.

`GETVAL` Return the value of a single semaphore.

`SETVAL` Set the value of a single semaphore. In this case, `arg` is taken as `arg.val`, an `int`.

`GETPID` Return the PID of the process that performed the last operation on the semaphore or array.

`GETNCNT` Return the number of processes waiting for the value of a semaphore to increase.

`GETZCNT` Return the number of processes waiting for the value of a particular semaphore to reach zero.

`GETALL` Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

`SETALL` Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

`IPC_STAT`
Return the status information contained in the control structure for the semaphore set, and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

`IPC_SET` Set the effective user/group identification and permissions In this case, `arg` is taken as `arg.buf`.

`IPC_RMID`
Remove the specified semaphore set.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID commands. Read/write permission is required as applicable for the other control commands.

The following program illustrates semctl().

Figure 3-16 *Sample Program to Illustrate* semctl()

```
/*
** semctl.c:   Illustrate the semctl() system call.
**
**   This is a simple exerciser of the semctl() system call.  It
** allows you to perform one control operation on one semaphore set.
** It gives up immediately if any control operation fails, so be careful not
** to set permissions to preclude read permission; you won't be able to reset
** the permissions with this code if you do.
*/

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>
#include  <time.h>

struct semid_ds semid_ds;

static void    do_semctl();
static void    do_stat();
extern char    *malloc();
extern void    exit();
extern void    perror();

char      warning_message[] = "If you remove read permission for\
 yourself, this program will fail frequently!";

main()
{
     union semun    arg; /* union to be passed to semctl() */
     int       cmd, /* command to be given to semctl() */
               i,   /* work area */
               semid,    /* semid to be passed to semctl() */
               semnum;   /* semnum to be passed to semctl() */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     (void) fprintf(stderr, "Enter desired semid value: ");
     (void) scanf("%i", &semid);

     (void) fprintf(stderr, "Valid semctl cmd values are:\n");
     (void) fprintf(stderr, "\tGETALL   =  %d\n", GETALL);
     (void) fprintf(stderr, "\tGETNCNT  =  %d\n", GETNCNT);
     (void) fprintf(stderr, "\tGETPID   =  %d\n", GETPID);
     (void) fprintf(stderr, "\tGETVAL   =  %d\n", GETVAL);
     (void) fprintf(stderr, "\tGETZCNT  =  %d\n", GETZCNT);
     (void) fprintf(stderr, "\tIPC_RMID =  %d\n", IPC_RMID);
     (void) fprintf(stderr, "\tIPC_SET  =  %d\n", IPC_SET);
     (void) fprintf(stderr, "\tIPC_STAT =  %d\n", IPC_STAT);
     (void) fprintf(stderr, "\tSETALL   =  %d\n", SETALL);
     (void) fprintf(stderr, "\tSETVAL   =  %d\n", SETVAL);
     (void) fprintf(stderr, "\nEnter desired cmd: ");
     (void) scanf("%i", &cmd);
```

```
        /* Perform some setup operations needed by multiple commands. */
        switch (cmd) {
        case GETVAL:
        case SETVAL:
        case GETNCNT:
        case GETZCNT:
             /* Get the semaphore number for these commands. */
             (void) fprintf(stderr, "\nEnter desired semnum value: ");
             (void) scanf("%i", &semnum);
             break;

        case GETALL:
        case SETALL:
             /* Allocate a buffer for the semaphore values. */
             (void) fprintf(stderr,
                  "Get number of semaphores in the set.\n");
             arg.buf = &semid_ds;
             do_semctl(semid, 0, IPC_STAT, arg);
             if (arg.array =
                  (ushort *)malloc((unsigned)
                       (semid_ds.sem_nsems * sizeof(ushort)))) {
                  /* Break out if we got what we needed. */
                  break;
             }
             (void) fprintf(stderr,
                  "semctl: unable to allocate space for %d values\n",
                  semid_ds.sem_nsems);
             exit(2);
             /*NOTREACHED*/
        }
        /* Get the rest of the arguments needed for the specified command. */
        switch (cmd) {
        case SETVAL:
             /* Set value of one semaphore. */
             (void) fprintf(stderr, "\nEnter desired semaphore value: ");
             (void) scanf("%i", &arg.val);
             do_semctl(semid, semnum, SETVAL, arg);
             /* Fall through to verify the result. */
             (void) fprintf(stderr,
                  "Perform semctl GETVAL command to verify results.\n");
        case GETVAL:
             /* Get value of one semaphore. */
             arg.val = 0;
             do_semctl(semid, semnum, GETVAL, arg);
             break;
        case GETPID:
             /* Get PID of last process to successfully complete a
                semctl(SETVAL), semctl(SETALL), or semop() on the
                semaphore. */
             arg.val = 0;
             do_semctl(semid, 0, GETPID, arg);
             break;
        case GETNCNT:
             /* Get number of processes waiting for semaphore value
                to increase. */
             arg.val = 0;
             do_semctl(semid, semnum, GETNCNT, arg);
             break;
        case GETZCNT:
             /* Get number of processes waiting for semaphore value
```

```
            to become zero. */
        arg.val = 0;
        do_semctl(semid, semnum, GETZCNT, arg);
        break;

    case SETALL:
        /* Set the values of all semaphores in the set. */
        (void) fprintf(stderr, "There are %d semaphores in the set.\n",
            semid_ds.sem_nsems);
        (void) fprintf(stderr, "Enter desired semaphore values:\n");
        for (i = 0; i < semid_ds.sem_nsems; i++) {
            (void) fprintf(stderr, "Semaphore %d: ", i);
            (void) scanf("%hi", &arg.array[i]);
        }
        do_semctl(semid, 0, SETALL, arg);

        /* Fall through to verify the results. */
        (void) fprintf(stderr,
            "Perform semctl GETALL command to verify results.\n");

    case GETALL:
        /* Get and print the values of all semaphores in the set.*/
        do_semctl(semid, 0, GETALL, arg);
        (void) fprintf(stderr, "The values of the %d semaphores are:\n",
            semid_ds.sem_nsems);
        for (i = 0; i < semid_ds.sem_nsems; i++)
            (void) fprintf(stderr, "%d ", arg.array[i]);
        (void) fprintf(stderr, "\n");
        break;

    case IPC_SET:
        /* Modify mode and/or ownership. */
        arg.buf = &semid_ds;
        do_semctl(semid, 0, IPC_STAT, arg);
        (void) fprintf(stderr, "Status before IPC_SET:\n");
        do_stat();

        (void) fprintf(stderr, "Enter desired sem_perm.uid value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.uid);

        (void) fprintf(stderr, "Enter desired sem_perm.gid value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.gid);

        (void) fprintf(stderr, "%s\n", warning_message);
        (void) fprintf(stderr,
            "Enter desired sem_perm.mode value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.mode);

        do_semctl(semid, 0, IPC_SET, arg);

        /* Fall through to verify changes. */
        (void) fprintf(stderr, "Status after IPC_SET:\n");

    case IPC_STAT:
        /* Get and print current status. */
        arg.buf = &semid_ds;
        do_semctl(semid, 0, IPC_STAT, arg);
        do_stat();
        break;

    case IPC_RMID:
        /* Remove the semaphore set. */
        arg.val = 0;
        do_semctl(semid, 0, IPC_RMID, arg);
        break;

    default:
        /* Pass unknown command to semctl. */
        arg.val = 0;
        do_semctl(semid, 0, cmd, arg);
```

```
            break;
        }
        exit(0);
        /*NOTREACHED*/
}
/*
**      Print indication of arguments being passed to semctl(), call semctl(),
** and report the results.
**      If semctl() fails, do not return; this example doesn't deal with
** errors, it just reports them.
*/
static void
do_semctl(semid, semnum, cmd, arg)
union semun    arg;
int       cmd,
          semid,
          semnum;
{
     register int    i;   /* work area */

     (void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ",
          semid, semnum, cmd);
     switch (cmd) {
     case GETALL:
          (void) fprintf(stderr, "arg.array = %#x)\n", arg.array);
          break;
     case IPC_STAT:
     case IPC_SET:
          (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
          break;
     case SETALL:
          (void) fprintf(stderr, "arg.array = [", arg.buf);
          for (i = 0;i < semid_ds.sem_nsems;) {
               (void) fprintf(stderr, "%d", arg.array[i++]);
               if (i < semid_ds.sem_nsems)
                    (void) fprintf(stderr, ", ");
          }
          (void) fprintf(stderr, "])\n");
          break;
     case SETVAL:
     default:
          (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
          break;
     }
     i = semctl(semid, semnum, cmd, arg);
     if (i == -1) {
          perror("semctl: semctl failed");
               exit(1);
               /* NOTREACHED */
     }
     (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
     return;
}
/*
**   Display contents of commonly used pieces of the status structure.
*/
static void
do_stat()
{
     (void) fprintf(stderr, "sem_perm.uid = %d\n", semid_ds.sem_perm.uid);
     (void) fprintf(stderr, "sem_perm.gid = %d\n", semid_ds.sem_perm.gid);
     (void) fprintf(stderr, "sem_perm.cuid = %d\n", semid_ds.sem_perm.cuid);
     (void) fprintf(stderr, "sem_perm.cgid = %d\n", semid_ds.sem_perm.cgid);
```

```
        (void) fprintf(stderr, "sem_perm.mode = %#o, ",
             semid_ds.sem_perm.mode);
        (void) fprintf(stderr, "access permissions = %#o\n",
             semid_ds.sem_perm.mode & 0777);
        (void) fprintf(stderr, "sem_nsems = %d\n", semid_ds.sem_nsems);
        (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
             ctime(&semid_ds.sem_otime) : "Not Set\n");
        (void) fprintf(stderr, "sem_ctime = %s", ctime(&semid_ds.sem_ctime));
}
```

Performing Semaphore Operations with `semop()`

The `semop()` system call is used to perform operations on a semaphore set. It's synopsis is as follows:

Figure 3-17 *Synopsis of* `semop()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(semid, sops, nsops)
int          semid;
struct       sembuf *sops;
unsigned     nsops;
```

The `semid` argument is the semaphore ID that was returned by a previous `semget()` call. The `sops` argument is a pointer to an array of structures, each of which contains the following information about a semaphore operation:

- The semaphore number.
- The operation to be performed.
- Control flags, if any.

`sembuf` is the structure of semaphores in the array, as defined in the `<sys/sem.h>` header file.

The `nsops` argument specifies the length of the array, the maximum size of which is determined by the `SEMOPM` configuration option; this is the maximum number of operations allowed by a single `semop()` call, 100 by default.

The operation to be performed is determined as follows:

- A positive integer means to increment the semaphore value by that amount.
- A negative integer means to increment the semaphore value by that amount. However, a semaphore can never take on a negative value. An attempt to set a semaphore to a value below zero either will either fail or block, depending on whether or not `IPC_NOWAIT` is in effect.
- A value of zero means to wait for the semaphore value to reach zero.

The following control flags can be used with `semop()`:

IPC_NOWAIT this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call will be unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.

SEM_UNDO this command allows individual operations in the array to be undone when the process exits.

The following program illustrates the semop() system call.

Figure 3-18 *Sample Program to Illustrate* semop()

```
/*
** semop.c:  Illustrate the semop() system call.
**
**   This is a simple exerciser of the semop() system call.  It allows
** you to set up arguments for semop(), make the call, and reports the
** results repeatedly on one semaphore set.  You must have read
** permission on the semaphore set or this exerciser will fail.  (It needs
** read permission to get the number of semaphores in the set and report
** their values before and after calls to semop().)
*/

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

static intask();
extern void    exit();
extern void    free();
extern char    *malloc();
extern void    perror();

static struct semid_ds   semid_ds; /* status of semaphore set */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
 semaphore values.  Giving up.\n";
static char    error_mesg2[] = "semop: Can't allocate space for %d\
 sembuf structures.  Giving up.\n";

main()
{
     register int   i;   /* work area */
     int       nsops;    /* number of operations to be performed */
     int       semid;    /* semid of semaphore set */
     struct sembuf  *sops;    /* ptr to operations to be performed */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     /* Loop until the invoker doesn't want to do anymore. */
     while (nsops = ask(&semid, &sops)) {

          /* Initialize the array of operations to be performed.*/
          for (i = 0; i < nsops; i++) {
               (void) fprintf(stderr,
```

```
                    "\nEnter desired values for operation %d of %d.\n",
                    i + 1, nsops);
               (void) fprintf(stderr,
                    "sem_num(valid values are 0 <= sem_num < %d): ",
                    semid_ds.sem_nsems);
               (void) scanf("%hi", &sops[i].sem_num);
               (void) fprintf(stderr, "sem_op: ");
               (void) scanf("%hi", &sops[i].sem_op);
               (void) fprintf(stderr,
                    "Expected flags in sem_flg are:\n");
               (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
                    IPC_NOWAIT);
               (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
                    SEM_UNDO);
               (void) fprintf(stderr, "sem_flg: ");
               (void) scanf("%hi", &sops[i].sem_flg);
          }

          /* Recap the call to be made. */
          (void) fprintf(stderr,
               "\nsemop: Calling semop(%d, &sops, %d) with:",
               semid, nsops);
          for (i = 0; i < nsops; i++)
          {
               (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
                    sops[i].sem_num);
               (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
               (void) fprintf(stderr, "sem_flg = %#o\n",
                    sops[i].sem_flg);
          }

          /* Make the semop() call and report the results. */
          if ((i = semop(semid, sops, nsops)) == -1) {
               perror("semop: semop failed");
          } else {
               (void) fprintf(stderr, "semop: semop returned %d\n", i);
          }
     }
     /*NOTREACHED*/
}

/*
**   Ask user if (s)he wants to continue.
**
** On the first call:
**   Get the semid to be processed and supply it to the caller.
** On each call:
**   1. Print current semaphore values.
**   2. Ask user how many operations are to be performed on next call to
**      semop.  Allocate an array of sembuf structures sufficient for the
**      job and set caller supplied pointer to that array.  (The array
**      is reused on subsequent calls as long as it is big enough.  If
**      it isn't big enough, it is freed and a larger array is allocated.)
*/
static
ask(semidp, sopsp)
int       *semidp;  /* pointer to semid (only used first time) */
struct sembuf  **sopsp;
{
     static union semun  arg;      /* argument to semctl */
     int            i;        /* work area */
     static int     nsops = 0;/* size of currently allocated
                              sembuf array */
     static int     semid = -1;    /* semid supplied by user */
```

```
        static struct sembuf*sops;          /* pointer to allocated array */

        if (semid < 0) {

             /* First call; get semid from user and the current state of
                the semaphore set. */
             (void) fprintf(stderr,
                  "Enter semid of the semaphore set you want to use: ");
             (void) scanf("%i", &semid);
             *semidp = semid;
             arg.buf = &semid_ds;
             if (semctl(semid, 0, IPC_STAT, arg) == -1) {
                  perror("semop: semctl(IPC_STAT) failed");

                  /* Note that if semctl fails, semid_ds remains filled with
                      zeroes, so later test for number of semaphores will be zero. */
                  (void) fprintf(stderr,
                      "Before and after values will not be printed.\n");
             } else {
                  if ((arg.array = (ushort *)malloc(
                      (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
                      == NULL) {
                       (void) fprintf(stderr, error_mesg1,
                            semid_ds.sem_nsems);
                       exit(1);
                  }
             }
        }

        /* Print current semaphore values. */
        if (semid_ds.sem_nsems) {
             (void) fprintf(stderr, "There are %d semaphores in the set.\n",
                  semid_ds.sem_nsems);
             if (semctl(semid, 0, GETALL, arg) == -1) {
                  perror("semop: semctl(GETALL) failed");
             } else {
                  (void) fprintf(stderr, "Current semaphore values are:");
                  for (i = 0; i < semid_ds.sem_nsems;
                       (void) fprintf(stderr, " %d", arg.array[i++]))
                       ;
                  (void) fprintf(stderr, "\n");
             }
        }

        /* Find out how many operations are going to be done in the next
           call and allocate enough space to do it. */
        (void) fprintf(stderr, "How many semaphore operations do you want %s\n",
             "on the next call to semop()?");
        (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
        i = 0;
        if (scanf("%i", &i) == EOF || i == 0)
             exit(0);
        if (i > nsops) {
             if (nsops)
                  free((char *)sops);
             nsops = i;
             if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
                  sizeof(struct sembuf)))) == NULL) {
                  (void) fprintf(stderr, error_mesg2, nsops);
                  exit(2);
             }
        }
        *sopsp = sops;
        return (i);
   }
```

3.5. Shared Memory

In the SunOS operating system, the most effecient method for implementing shared memory applications is to rely on native virtual memory management and the `mmap`(2) system call. For shared memory applications that are to be compatible with System V, the SunOS operating system also provides the standard System V shared memory facilities.

Shared memory allows more than one process at a time to attach a segment of physical memory to its virtual address space. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

Using System V shared memory, a process creates a shared memory segment using the `shmget`(2) system call. This call can also be used to obtain the ID of an existing shared segment. The creating process sets the permissions, and the size in bytes for the segment.

The original owner/creator of a shared memory segment can assign ownership to another user with the `shmctl`(2) system call; it can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`.

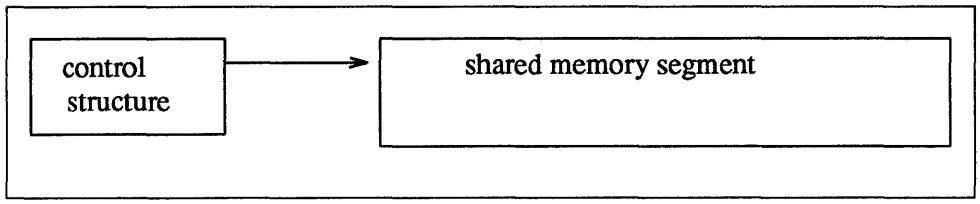
Once created, a shared segment can be attached to a process's address space using the `shmat()` system call; it can be detached using `shmdt()`. (See `shmop`(2) for details.) The attaching process must have the appropriate permissions for `shmat()` to succeed. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment may be attached multiple times by the same process.

If any of the above-mentioned system calls fails, it returns –1, and sets the external variable `errno` to the appropriate value.

Structure of a Shared Memory Segment

A shared memory segment is composed of a control structure with a unique ID that points to an area of physical memory. The identifier for the segment is referred to as the `shmid`.

Figure 3-19 *Structure of a Shared Memory Segment*



The data structure includes the following information about the memory segment:

- Access permissions.
- Segment size.
- The PID of the process performing last operation.

- The PID of the creator process.
- The current number of processes to which the segment is attached.
- The time of the last attachment.
- The time of the last detachment.
- The time of the last change to the segment.
- Memory map segment descriptor pointer.

The structure definition for the shared memory segment control structure can be found in `<sys/shm.h>`. This structure definition is shown below.

```
/*
 *  There is a shared mem id data structure for each segment in the system.
 */

struct shmid_ds {
    struct  ipc_perm shm_perm;  /* operation permission struct */
    uint    shm_segsz;          /* size of segment in bytes */
    ushort  shm_lpid;           /* pid of last shmop */
    ushort  shm_cpid;           /* pid of creator */
    ushort  shm_nattch;         /* number of current attaches */
    time_t  shm_atime;          /* last shmat time */
    time_t  shm_dtime;          /* last shmdt time */
    time_t  shm_ctime;          /* last change time */
    struct  anon_map *shm_amp;  /* segment anon_map pointer */
};
```

Note that the `shm_perm` member of this structure uses `ipc_perm` as a template, as defined in `<sys/ipc.h>`.

Using `shmget()` to Get Access to a Shared Memory Segment

The `shmget()` system call is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (`shmid`). When it fails, it returns –1, and sets `errno` to the appropriate error code. `shmget()` has the following synopsis:

Figure 3-20 *Synopsis of* `shmget()`

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int shmget(key, size, shmflg)
key_t   key;
int     size, shmflg;
```

The value passed as the `shmflg` argument must be an integer, which incorporates settings for the segment's permissions and control flags, as described under *System V IPC Permissions*, above.

The SHMMNI system configuration option determines the maximum number of shared memory segments that are allowed, 100 by default.

The system call will fail if the size value is less than SHMMIN or greater than SHMMAX, the configuration options for the minimum and maximum segment sizes. By default, SHMIN is 1, SHMAX is 1048576.

The following sample program illustrates the shmget() system call.

Figure 3-21 *Sample Program to Illustrate* shmget()

```
/*
** shmget.c: Illustrate the shmget() system call.
**
**   This is a simple exerciser of the shmget() system call.
** It prompts for the arguments, makes the call, and reports the results.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern void    exit();
extern void    perror();

main()
{
     key_tkey; /* key to be passed to shmget() */
     int  shmflg;   /* shmflg to be passed to shmget() */
     int  shmid;    /* return value from shmget() */
     int  size;/* size to be passed to shmget() */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     /* Get the key. */
     (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
     (void) fprintf(stderr, "Enter desired key: ");
     (void) scanf("%li", &key);

     /* Get the size of the segment. */
     (void) fprintf(stderr, "Enter desired size: ");
     (void) scanf("%i", &size);

     /* Get the shmflg value. */
     (void) fprintf(stderr, "Expected flags for the shmflg argument are:\n");
     (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
     (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
     (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
     (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
     (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
     (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
     (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
     (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
     (void) fprintf(stderr, "Enter desired shmflg: ");
     (void) scanf("%i", &shmflg);

     /* Make the call and report the results. */
     (void) fprintf(stderr, "shmget: Calling shmget(%#lx, %d, %#o)\n",
          key, size, shmflg);
     if ((shmid = shmget (key, size, shmflg)) == -1) {
```

```
            perror("shmget: shmget failed");
            exit(1);
        } else {
            (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
            exit(0);
        }
        /*NOTREACHED*/
    }
```

Controlling a Shared Memory Segment with `shmctl()`

The `shmctl()` system call is used to alter the permissions and other characteristics of a shared memory segment. It synopsis is as follows:

Figure 3-22 *Synopsis of* `shmctl()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int     shmctl(shmid, cmd, buf)
int     shmid, cmd;
struct  shmid_ds *buf;
```

The `shmid` argument is the ID of the shared memory segment as returned by `shmget()`. The `cmd` argument is one of following control commands:

`SHM_LOCK`
: Lock the specified shared memory segment in memory. The process must have effective ID of super-user to perform this command.

`SHM_UNLOCK`
: Unlock the shared memory segment. The process must have effective ID of super-user to perform this command.

`IPC_STAT`
: Return the status information contained in the control structure, and place it in the buffer pointed to by `buf`. The process must have read permission on the segment to perform this command.

`IPC_SET`
: Set the effective user and group identification, and access permissions. The process must have an effective ID of owner, creator or super-user to perform this command.

`IPC_RMID`
: Remove the shared memory segment. The process must have an effective ID of owner, creator or super-user to perform this command.

The example program below allows you to illustrate `shmctl()`.

Figure 3-23 *Sample Program to Illustrate* `shmctl()`

```
/*
** shmctl.c: Illustrate the shmctl() system call.
**
**   This is a simple exerciser of the shmctl() system call.  It allows
** you to perform one control operation on one shared memory segment.
** (Some operations are done for the user whether requested or not.  It gives
** up immediately if any control operation fails.  Be careful not to set
** permissions to preclude read permission; you won't be able to reset the
** permissions with this code if you do.)
*/

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <time.h>

static void    do_shmctl();
extern void    exit();
extern void    perror();

main()
{
     int       cmd;
     int       shmid;
     struct shmid_dsshmid_ds;

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");

     /* Get shmid and cmd. */
     (void) fprintf(stderr, "Enter the shmid for the desired segment: ");
     (void) scanf("%i", &shmid);

     (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
     (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
     (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
     (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
     (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
     (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
     (void) fprintf(stderr, "Enter the desired cmd value: ");
     (void) scanf("%i", &cmd);

     switch (cmd) {
     case IPC_STAT:
          /* Get shared memory segment status. */
          break;

     case IPC_SET:
          /* Set owner UID and GID and permissions. */
          /* Get and print current values. */
          do_shmctl(shmid, IPC_STAT, &shmid_ds);

          /* Set UID, GID, and permissions to be loaded. */
          (void) fprintf(stderr, "\nEnter desired shm_perm.uid: ");
          (void) scanf("%hi", &shmid_ds.shm_perm.uid);
          (void) fprintf(stderr, "Enter desired shm_perm.gid: ");
          (void) scanf("%hi", &shmid_ds.shm_perm.gid);
          (void) fprintf(stderr,
               "Note: Keep read permission for yourself.\n");
          (void) fprintf(stderr, "Enter desired shm_perm.mode: ");
          (void) scanf("%hi", &shmid_ds.shm_perm.mode);
```

```
            break;
        case IPC_RMID:
            /* Remove the segment when the last attach point is detached. */
            break;
        case SHM_LOCK:
            /* Lock the shared memory segment. */
            break;
        case SHM_UNLOCK:
            /* Unlock the shared memory segment. */
            break;
        default:
            /* Unknown command will be passed to shmctl. */
            break;
        }
        do_shmctl(shmid, cmd, &shmid_ds);
        exit(0);
        /*NOTREACHED*/
}
/*
**   Display the arguments being passed to shmctl(), call shmctl(), and
** report the results.
**   If shmctl() fails, do not return; this example doesn't deal with
** errors, it just reports them.
*/
static void
do_shmctl(shmid, cmd, buf)
int         shmid,
            cmd;
struct shmid_ds *buf;
{
        register int    rtrn;/* hold area */
        (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n",
            shmid, cmd);
        if (cmd == IPC_SET) {
            (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
                buf->shm_perm.uid);
            (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
                buf->shm_perm.gid);
            (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
                buf->shm_perm.mode);
        }
        if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
            perror("shmctl: shmctl failed");
            exit(1);
        } else {
            (void) fprintf(stderr, "shmctl: shmctl returned %d\n", rtrn);
        }
        if (cmd != IPC_STAT && cmd != IPC_SET)
            return;
        /* Print the current status. */
        (void) fprintf(stderr, "\nCurrent status:\n");
        (void) fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
        (void) fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
        (void) fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
        (void) fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
        (void) fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
        (void) fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.key);
        (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
        (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
        (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
```

```
        (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
        (void) fprintf(stderr, "\tshm_atime = %s",
            buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
        (void) fprintf(stderr, "\tshm_dtime = %s",
            buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
        (void) fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
    }
```

Attaching and Detaching a Shared Memory Segment with `shmat()` and `shmdt()`

`shmat()` and `shmdt()` are used to attach and detach shared memory segments. Their synopses are as follows:

Figure 3-24 *Synopses of* `shmat()` *and* `shmdt()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat(shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt(shmaddr)
char *shmaddr;
```

Upon successful completion, the `shmat()` system call returns a pointer to the head of the shared segment; when unsuccessful, it returns '`(char *)-1`' and sets the external variable `errno` to the appropriate error code.

The `shmid` argument is the ID of an existing shared memory segment. The `shmaddr` argument is the address at which to attach the segment. If supplied as zero, the system provides a suitable address. For the sake of portability, it is usually better to allow the system to determine the address.

The `shmflg` argument is a control flag used to pass the `SHM_RND` and `SHM_RDONLY` flags to the `shmat()` system call.

The `shmdt()` system call detaches the shared memory segment located at the address indicated by `shmaddr`. Upon successful completion, `schmdt()` returns zero; when unsuccessful, it returns –1 and sets the external variable `errno` to the appropriate error code.

The following sample program illustrates `shmat()` and `shmdt()`.

Figure 3-25 *Sample Program to Illustrate* `shmat()` *and* `shmdt()`

```
/*
** shmop.c: Illustrate the shmat() and shmdt() system calls.
**
**   This is a simple exerciser for the shmat() and shmdt() system
** calls.  It allows you to attach and detach segments and to
** write strings into and read strings from attached segments.
*/

#include  <stdio.h>
#include  <setjmp.h>
#include  <signal.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

#define   MAXnap    4    /* Maximum number of concurrent attaches. */

static          ask();
static void     catcher();
extern void     exit();
static          good_addr();
extern void     perror();
extern char     *shmat();

static struct state {    /* Internal record of currently attached segments. */
     int  shmid;         /* shmid of attached segment */
     char *shmaddr; /* attach point */
     int  shmflg;        /* flags used on attach */
}    ap[MAXnap];     /* State of current attached segments. */

static intnap; /* Number of currently attached segments. */
static jmp_buf segvbuf;  /* Process state save area for SIGSEGV catching. */

main()
{
     register int        action;   /* action to be performed */
     char           *addr;    /* address work area */
     register int        i;   /* work area */
     register struct state    *p;  /* ptr to current state entry */
     void           (*savefunc)(); /* SIGSEGV state hold area */

     (void) fprintf(stderr,
          "All numeric input is expected to follow C conventions:\n");
     (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
     (void) fprintf(stderr, "\t0...  is interpreted as octal,\n");
     (void) fprintf(stderr, "\totherwise, decimal.\n");
     while (action = ask()) {
          if (nap) {
               (void) fprintf(stderr,
                    "\nCurrently attached segment(s):\n");
               (void) fprintf(stderr, " shmid    address\n");
               (void) fprintf(stderr, "------ ----------\n");
               p = &ap[nap];
               while (p-- != ap) {
                    (void) fprintf(stderr, "%6d", p->shmid);
                    (void) fprintf(stderr, "%#11x", p->shmaddr);
                    (void) fprintf(stderr, " Read%s\n",
                         (p->shmflg & SHM_RDONLY) ?
                         "-Only" : "/Write");
               }
          } else
               (void) fprintf(stderr,
                    "\nNo segments are currently attached.\n");
```

```
switch (action) {
case 1:   /* Shmat requested. */
     /* Verify that we have space for another attach. */
     if (nap == MAXnap) {
          (void) fprintf(stderr, "%s %d %s\n",
               "This simple example will only allow",
               MAXnap, "attached segments.");
          break;
     }
     p = &ap[nap++];

     /* Get the arguments, make the call, report the
        results, and update the current state array. */
     (void) fprintf(stderr,
          "Enter shmid of segment to attach: ");
     (void) scanf("%i", &p->shmid);

     (void) fprintf(stderr, "Enter desired shmaddr: ");
     (void) scanf("%i", &p->shmaddr);

     (void) fprintf(stderr,
          "Meaningful shmflg values are:\n");
     (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
          SHM_RDONLY);
     (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
          SHM_RND);
     (void) fprintf(stderr, "Enter desired shmflg value: ");
     (void) scanf("%i", &p->shmflg);

     (void) fprintf(stderr,
          "shmop: Calling shmat(%d, %#x, %#o)\n",
          p->shmid, p->shmaddr, p->shmflg);
     p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
     if(p->shmaddr == (char *)-1)  {
          perror("shmop: shmat failed");
          nap--;
     } else {
          (void) fprintf(stderr,
               "shmop: shmat returned %#8.8x\n",
               p->shmaddr);
     }
     break;

case 2:   /* Shmdt requested. */
     /* Get the address, make the call, report the results,
        and make the internal state match. */
     (void) fprintf(stderr,
          "Enter desired detach shmaddr: ");
     (void) scanf("%i", &addr);

     i = shmdt(addr);
     if(i == -1)  {
          perror("shmop: shmdt failed");
     } else {
          (void) fprintf(stderr,
               "shmop: shmdt returned %d\n", i);
          for (p = ap, i = nap; i--; p++) {
               if (p->shmaddr == addr)
                    *p = ap[--nap];
          }
     }
     break;

case 3:   /* Read from segment requested. */
     if (nap == 0)
          break;

     (void) fprintf(stderr, "Enter address of an %s",
```

```
                    "attached segment: ");
                (void) scanf("%i", &addr);

                if (good_addr(addr))
                    (void) fprintf(stderr, "String @ %#x is '%s'\n",
                        addr, addr);
                break;

            case 4:   /* Write to segment requested. */
                if (nap == 0)
                    break;

                (void) fprintf(stderr, "Enter address of an %s",
                    "attached segment: ");
                (void) scanf("%i", &addr);

                /* Set up SIGSEGV catch routine to trap attempts to
                   write into a read-only attached segment. */
                savefunc = signal(SIGSEGV, catcher);

                if (setjmp(segvbuf)) {
                    (void) fprintf(stderr, "shmop: %s: %s\n",
                        "SIGSEGV signal caught",
                        "Write aborted.");
                } else {
                    if (good_addr(addr)) {
                        (void) fflush(stdin);
                        (void) fprintf(stderr, "%s %s %#x:\n",
                            "Enter one line to be copied",
                            "to shared segment attached @",
                            addr);
                        (void) gets(addr);
                    }
                }
                (void) fflush(stdin);

                /* Restore SIGSEGV to previous condition. */
                (void) signal(SIGSEGV, savefunc);
                break;
            }
        }
        exit(0);
        /*NOTREACHED*/
}

/*
**  Ask for next action.
*/
static
ask()
{
        int  response; /* user response */

        do {
            (void) fprintf(stderr, "Your options are:\n");
            (void) fprintf(stderr, "\t^D = exit\n");
            (void) fprintf(stderr, "\t 0 = exit\n");
            (void) fprintf(stderr, "\t 1 = shmat\n");
            (void) fprintf(stderr, "\t 2 = shmdt\n");
            (void) fprintf(stderr, "\t 3 = read from segment\n");
            (void) fprintf(stderr, "\t 4 = write to segment\n");
            (void) fprintf(stderr,
                "Enter the number corresponding to your choice: ");

            /* Preset response so "^D" will be interpreted as exit. */
            response = 0;
            (void) scanf("%i", &response);
        } while (response < 0 || response > 4);
```

```
        return (response);
}

/*
**   Catch signal caused by attempt to write into shared memory segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
        longjmp(segvbuf, 1);
        /*NOTREACHED*/
}

/*
**   Verify that given address is the address of an attached segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
        register struct state    *p;  /* ptr to state of attached segment */

        for (p = ap; p != &ap[nap]; p++)
                if (p->shmaddr == address)
                        return(1);
        return(0);
}
```

4

SCCS — Source Code Control System

4.1. Introduction

Coordinating write access to source files is important when changes may be made by several people. Maintaining a record of updates allows you to determine when and why changes were made.

The Source Code Control System (SCCS) allows you to control write access to source files, and to monitor changes made to those files. SCCS allows only one user at a time to update a file, and records all changes in a *history* file.

SCCS allows you to:

- Retrieve copies of any version of the file from the SCCS history.
- Check out and lock a version of the file for editing, so that only you may make changes to it. SCCS prevents one user from unwittingly "clobbering" changes made by another.
- Check in your updates to the file. When you check in a file, you can also supply comments that summarize your changes.
- Back out changes made to your checked-out copy.
- Inquire about the availability of a file for editing.
- Inquire about differences between selected versions.
- Display the *version log* summarizing the changes checked in so far.

The `sccs` Command

The Source Code Control System is composed of the `sccs`(1) command, which is a front end for the utility programs in the `/usr/sccs` directory. The SCCS utility programs are listed under *Reference Tables*, at the end of this chapter.

Initializing the SCCS History File: `sccs create`

The `sccs create` command places your file under SCCS control. It creates a new history file, and uses the complete text of your source file as the initial version. By default, the history file resides in the `SCCS` subdirectory; you may have to create this subdirectory if it is not already present:

```
hermes% mkdir SCCS
hermes% sccs create program.c

program.c:
1.1
87 lines
```

The output from SCCS tells you the name of the "created" file, its version number (1.1), and the count of lines.

To prevent the accidental loss or damage to an original, `sccs create` makes a second link to it, prefixing the new filename with a comma (referred to as the "*comma-file.*") When the history file has been initialized successfully, SCCS retrieves a new, read-only version. Once you have verified the version against its comma-file, you can remove that file.

```
hermes% cmp ,program.c program.c
(no output means that the files match exactly)
hermes% rm ,program.c
```

Do not try to edit the read-only version that SCCS retrieves. Before you can edit the file, you must check it out using the `sccs edit` command described below.

To distinguish the history file from a current version, SCCS uses the '`s.`' prefix.

```
hermes% ls SCCS
s.program.c
```

Owing to this prefix, the history file is often referred to as the `s.` file ("*s-dot-file*"). For historical reasons, it may also be referred to as the "*SCCS-file.*"

The format of an SCCS history file is described in `sccsfile`(5).

Basic `sccs` Subcommands

The following `sccs` subcommands perform the basic version-control functions. They are summarized here, and, except for `create`, are described in detail under `sccs` *Subcommands*, below.

`create` Initialize the history file and first version, as described above.

`edit` Check out a writable version (for editing). SCCS retrieves a writable copy with you as the owner, and places a lock on the history file so that no one else can check in changes.

`delta` Check in your changes. This is the complement to the `sccs edit` operation. Before recording your changes, SCCS prompts for a comment, which it then stores in the history file's version log.

`get` Retrieve a read-only copy of the file from the `s.` file. By default, this is the most recent version. While the retrieved version can be used as a source file for compilation, formatting, or display, it is *not*

intended to be edited or changed in any way. (Attempting to bend the rules by changing permissions of a read-only version can result in your changes being lost.)

If you give a directory as a filename argument, `sccs` attempts to perform the subcommand on each `s.` file in that directory. Thus, the command:

```
sccs get SCCS
```

retrieves a read-only version for every `s.` file in the `SCCS` subdirectory.

`prt`
Display the version log, including comments associated with each version.

Deltas and Versions

When you check in a version, SCCS records only the line-by-line differences between the text you check in and the previous version. This set of differences is known as a *delta*. The version that is retrieved by an `edit` or `get` is constructed from the accumulated deltas checked in so far. The terms "delta" and "version" are often used synonymously. However, their meanings aren't exactly the same; it is possible to retrieve a version that omits selected deltas (see *Excluding Deltas from a Retrieved Version*, below).

SIDs

An SCCS delta ID, or SID, is the number used to represent a specific delta. This is a two-part number, with the parts separated by a dot (`.`). The SID of the initial delta is `1.1` by default. The first part of the SID is referred to as the *release* number, and the second, the *level* number. When you check in a delta, the level number is incremented automatically. The release number can be incremented as needed. SCCS also recognizes two additional fields for *branch* deltas (described under *Branch Deltas*, below).

Strictly speaking, an SID refers directly to a delta. However, it is often used to indicate the version constructed from a delta and its predecessors.

ID Keywords

SCCS recognizes and expands certain keywords in a source file, which you can use to include version-dependent information (such as the SID) into the text of the checked-in version. When the file is checked out for editing, ID keywords take the following form:

`%`*C*`%`

where *C* is a capital letter. When you check in the file, SCCS replaces the keywords with the information they stand for. For example, `%I%` expands to the SID of the current version.

You would typically include ID keywords either in a comment or in a string definition. If you do not include at least one ID keyword in your source file, SCCS issues the diagniostic:

```
No Id Keywords (cm7)
```

For more information about ID keywords, refer to *Incorporating ID Keywords*, below.

4.2. sccs Subcommands

Checking Files In and Out

The following subcommands are useful when retrieving versions or checking in changes.

Checking Out a File for Editing: sccs edit

To edit a source file, you must check it out first using `sccs edit`.[1] SCCS responds with the delta ID of the version just retrieved, and the delta ID it will assign when you check in your changes.

```
hermes% sccs edit program.c
1.1
new delta 1.2
87 lines
```

You can then edit it using a text editor.

If a writable copy of the file is present, `sccs edit` issues an error message; it does not overwrite the file if anyone has write access to it.

Checking in a New Version: sccs delta

Having first checked out your file and completed your edits, you can check in the changes using `sccs delta`.

Checking a file in is also referred to as "making a delta." Before checking in your updates, SCCS prompts you for comments. These typically include a brief summary of your changes.

```
hermes% sccs delta program.c
comments?
```

You can extend the comment to an additional input line by preceding the NEW-LINE with a backslash:

```
hermes% sccs delta program.c
comments? corrected typo in widget(), \
null pointer in n_crunch()
1.2
5 inserted
3 deleted
84 unchanged
```

[1] The `sccs edit` command is equivalent to using the `-e` option to `sccs get`.

Changed lines count as lines deleted and inserted.

SCCS responds by noting the SID of the new version, and the numbers of lines inserted, deleted and unchanged. SCCS removes the working copy. You can retrieve a read-only version using `sccs get`.

Think ahead before checking in a version. Making deltas after each minor edit can become excessive. On the other hand, leaving files checked out for so long that you forget about them can inconvenience others.

Comments should be meaningful, since you may return to the file one day.

It is important to check in all changed files before compiling or installing a module for general use. A good technique is to `edit` the files you need, make all necessary changes and tests, compile and debug the files until you are satisfied, check them in, retrieve read-only copies with get, and then recompile the module.

Retrieving a Version: `sccs get`

To get the most recent version of a file, use the command:

```
sccs get filename
```

For example:

```
hermes% sccs get program.c
1.2
86 lines
```

retrieves `program.c`, and reports the version number and the number of lines retrieved. The retrieved copy of `program.c` has permissions set to read-only.

Do not change this copy of the file, since SCCS will *not* create a new delta unless the file has been checked out. If you force changes into the retrieved copy, you may lose them the next time someone performs an `sccs get` or an `sccs edit` on the file.

Reviewing Pending Changes: `sccs diffs`

Changes made to a checked-out version, but which are not yet checked in, are said to be *pending*. When editing a file, you can find out what your pending changes are using '`sccs diffs`'. The `diffs` subcommand uses `diff`(1) to compare your working copy with the most recently checked-in version.

```
hermes% sccs diffs program.c
------ program.c ------
37c37
<       if (((cmd_p - cmd) + 1) == l_lim) {
---
>       if (((cmd_p - cmd) - 1) == l_lim) {
```

Most of the options to `diff` can be used. To invoke the `-c` option to `diff`, use the '`-C`' argument to '`sccs diffs`'.

Deleting Pending Changes: `sccs unedit`

`sccs unedit` backs out pending changes. This comes in handy if you damage the file while editing it and want to start over. `unedit` removes the checked-out version, unlocks the history file, and retrieves a read-only copy of the most recent version checked in. After using `unedit`, it is as if you hadn't checked out the file at all. To resume editing, use `sccs edit` to check the file out again. (See also, *Repairing a Writable Copy*, below.)

Combining `delta` and `get`: `sccs delget`

`sccs delget` combines the actions of `delta` and `get`: it checks in your changes and then retrieves a read-only copy of the new version. However, if SCCS encounters an error during the `delta`, it does not perform the `get`. When processing a list of filenames, `delget` applies all the `deltas` it can, and if errors occur, omits all of the `gets`.

Combining `delta` and `edit`: `sccs deledit`

`sccs deledit` performs a `delta` followed by an `edit`. You can use this to check in a version and immediately resume editing.

Retrieving a Version by SID: `sccs get -r`

The `-r` option allows you to specify the SID to retrieve:

```
hermes% sccs get -r1.1 program.c
1.1
87 lines
```

Retrieving a Version by Date and Time: `sccs get -c`

In some cases you don't know the SID of the delta you want, but you do know the date on (or before) which it was checked in. You can retrieve the latest version checked in before a given date and time using the `-c` option and a date-time argument of the form:

`-c`*yy* [*mm* [*dd* [*hh* [*mm* [*ss*]]]]]

For example:

```
hermes% sccs get -c880722120000 program.c
1.2
86 lines
```

retrieves whatever version was current as of July 22, 1988 at 12:00 noon. Trailing fields can be omitted (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be written as:

```
sccs get -c"88/07/22 12:00:00" program.c
```

Repairing a Writable Copy: `sccs get -k -G`

Without checking out a new version, `sccs get -k -G`*filename* retrieves a writable copy of the text, and places it in the file specified by '`-G`'. This can be useful when you want to replace or repair a damaged working copy using `diff` and your favorite editor.

Incorporating Version-Dependent Information: ID Keywords

As mentioned above, SCCS allows you to include version-dependent information in a checked-in version through the use of *ID keywords*. These keywords, which you insert in the file, are automatically replaced by the corresponding information when you check in your changes. SCCS ID keywords take the form:

```
%C%
```

where *C* is an upper case letter.

For instance, `%I%` expands to the SID of the most recent delta. `%W%` includes the filename, the SID, and the unique string `@(#)` into the file. This string is searched for by the `what` command in both text and binary files (allowing you to see which source versions a file or program was built from). The `%G%` keyword expands to the date of the latest delta. Other ID keywords and the strings they expand to are listed in the *Identification Keywords*, table under *Reference Tables* at the end of this chapter.

To include version dependent information in a C program, you can use a line like this:

```
static char SccsId[ ] = "%W%\t%G%";
```

Defining a string in this way allows version information to be compiled into the C object file. If you use this technique to put ID keywords into header (`.h`) files, use a different variable in each header file. This prevents errors from attempts to redefine the (static) variables.

If the file were named `program.c`, this line would expand to the following when version 1.2 is retrieved:

```
static char SccsId[ ] = "@(#)program.c  1.2  08/29/80";
```

Since the string is defined in the compiled program, this technique allows you to include source-file information within the compiled program, which the `what` command can report:

```
hermes% cd /usr/ucb
hermes% what sccs
sccs
        sccs.c 1.13 88/02/08 SMI
```

For shell scripts and the like, you can include ID keywords within comments:

```
#       %W%     %G%
...
```

If you check in a version containing *expanded* keywords, the version-dependent information will no longer be updated. To alert you to this situation, SCCS gives you the warning:

```
No Id Keywords (cm7)
```

when a `get`, `edit`, or `create` finds no ID keywords.

Making Inquiries

The following subcommands are useful for inquiring about the status of a file or its history.

Seeing Which Version Has Been Retrieved: The `what` Command

Since SCCS allows you (or others) to retrieve any version in the file's history, there is no guarantee that a working copy present in the directory reflects the version you desire. The `what` command scans files for SCCS ID keywords. It also scans binary files for keywords, allowing you to see which source versions a program was compiled from.

```
hermes% what program.c program
program.c:
        program.c 1.1 88/07/05 SMI;
program:
        program.c 1.1 88/07/05 SMI;
```

In this case, the file contains a working copy of version 1.1.

Determining the Most Recent Version: `sccs get -g`

To see the SID of the latest delta, you can use `sccs get -g`:

```
hermes% sccs get -g program.c
1.2
```

In this case, the most recent delta is 1.2. Since this is more recent than the version reflected by `what` in the example above, you would probably want to `get` the new version.

Determining Who Has a File Checked Out: `sccs info`

To find out what files are being edited, type:

```
sccs info
```

This subcommand displays a list of all the files being edited, along with other information, such as the name of the user who checked the file out. Similarly, you can use

```
sccs check
```

silently returns a non-zero exit status if anything is being edited. This can be used within a makefile to force `make`(1) to halt if it should find that a source file is checked out.

If you know that all the files that you have checked out are ready to be checked in, you can use:

```
sccs delta 'sccs tell -u'
```

to process them all. `tell` lists only the names of files being edited, one per line. With the `-u` option, `tell` reports only those files checked out to you. If you supply a username as an argument to `-u`, `sccs tell` reports only the files checked out to that user.

Displaying Delta Comments: `sccs prt`

`sccs prt` produces a listing of the version log, also referred to as the *delta table*, which includes the SID, time and date of creation, and the name of the user who checked in each version, along with the number of lines inserted, deleted, and unchanged, and the commentary:

```
hermes% sccs prt program.c
D 1.2   80/08/29 12:35:31       pers    2       1       00005/00003/00084
corrected typo in widget(),
null pointer in n_crunch()

D 1.1   79/02/05 00:19:31       zeno    1       0       00087/00000/00000
date and time created 80/06/10 00:19:31 by zeno
```

To display only the most recent entry, use the `-y` option.

Updating a Delta Comment: `sccs cdc`

If you forget to include something important in a comment, you can add the missing information using

```
sccs cdc -rsid
```

The delta must be the most recent (or the most recent in its branch, see *Branches*, below), and you must either be the user who checked the delta in, or you must own and have permission to write on both the history file and the `SCCS` subdirectory. When you use `cdc`, SCCS prompts for your comments and inserts the new comment you supply:

```
hermes% sccs cdc -r1.2 program.c
comments? also taught get_in() to handle control chars
```

The new commentary, as displayed by `prt`, looks like:

```
hermes% sccs prt program.c
D 1.2   80/08/29 12:35:31       pers    2       1       00005/00003/00084
also taught get_in() to handle control chars
*** CHANGED *** 88/08/02 14:54:45 pers
corrected typo in widget(),
null pointer in n_crunch()

D 1.1   79/02/05 00:19:31       zeno    1       0       00087/00000/00000
date and time created 80/06/10 00:19:31 by zeno
```

Comparing Checked-In Versions: `sccs sccsdiff`

To compare two checked-in versions, use:

```
hermes% sccs sccsdiff -r1.1 -r1.2 program.c
```

to see the differences between delta 1.1 and delta 1.2. Most options to `diff` can be used. To invoke the `-c` option to `diff`, use the '`-C`' argument to '`sccsdiff`'. Instead of `-r`, you can use the `-c`*date-time* option to `sccs`.

Displaying the Entire History: `sccs get -m -p`

If you wish to see a listing of all changes made to the file and the delta in which each was made, you can use the `-m` and `-p` options to `get`:

```
hermes% sccs get -m -p program.c
1.2
84 lines
1.2     #define L_LEN 256
1.1
1.1     #include <stdio.h>
1.1
...
```

To find out what lines are associated with a particular delta, you can pipe the output through `grep`(1V):

```
sccs get -m -p program.c | grep '^1.2'
```

You can also use `-p`, by itself to send the retrieved version to the standard output, rather than to the file.

Creating Reports: `sccs prs -d`

You can use the `prs` subcommand with the `-d`*dataspec* option to derive reports about files under SCCS control. The *dataspec* argument offers a rich set of *i"data*keywords" that correspond to portions of the history file. Data keywords take the form:

:*X*:

and are listed in the *Data Keywords* table under *Reference Tables* at the end of this chapter. There is no limit on the number of times a data keyword may appear in the *dataspec* argument. A valid *dataspec* argument is a (quoted) string consisting of text and data keywords.

`prs` replaces each recognized keyword with the appropriate value from the history file.

The format of a data keyword value is either simple, in which case the expanded value is a simple string, or multi-line, in which case the expansion includes RETURN characters.

A TAB is specified by '`\t`' and a RETURN by '`\n`'.

Here are some examples:

```
hermes% sccs prs -d"Users and/or user IDs for :F: are:\n:UN:" program.c
Users and/or user IDs for s.program.c are:
zeno
pers

hermes% sccs prs -d"Newest delta for :M:: :I:.  Created :D: by :P:." -r program.c
Newest delta for program.c: 1.3.  Created 88/07/22 by zeno.
```

Deleting Committed Changes

Replacing a Delta: `sccs fix`

From time to time a delta is checked in that contains small bugs, such as typos, that need correcting but that do not require entries in the file's audit trail. Or, perhaps the comment for a delta is incomplete or in error, even when the text is correct. In either case, you can make additional updates and replace the version log entry for the most recent delta using `sccs fix`:

```
hermes% sccs fix -r1.2 program.c
```

This checks out version 1.2 of `program.c`. When you check the file back in, the current changes will replace delta 1.2 in the history file, and SCCS will prompt for a (new) comment. You must supply an SID with '`-r`'. Also, the delta that is specified must be a leaf (most recent) delta.

Although the previously-checked-in delta 1.2 is effectively deleted, SCCS retains a record of it, marked as deleted, in the history file.

Before using `sccs fix` it is a good idea to make a copy of the current version, just in case.

Removing a Delta: `sccs rmdel`

To remove all traces of the most recent delta, you can use the `rmdel` subcommand. You must specify the SID using `-r`. In most cases, using `fix` is preferable to `rmdel`, since `fix` preserves a record of "deleted" delta, while `rmdel` does not.[2]

Reverting to an Earlier Version

To retrieve a writable copy of an earlier version, use '`get -k`'. This can come in handy when you need to backtrack past several deltas.

To use an earlier delta as the basis for creating a new one:

- Check out the file as you normally would (using `sccs edit`).
- Retrieve a writable copy of an earlier "good" version (giving it a different filename) using `get -k`:

  `sccs get -k -r`*sid* `-G`*oldname filename*

  The `-G`*filename* option specifies the name of the newly retrieved version.
- Replace the current version with the older "good" version:

  `mv` *oldname filename*

- And finally, check the file back in. In some cases, it may be simpler just to exclude certain deltas. Or, refer to *Branch Deltas*, below, for information on how to use SCCS to manage divergent sets of updates to a file.

[2] Refer to `sccs-rmdel`(1) for more information.

Excluding Deltas from a Retrieved Version

Suppose that the changes that were made in delta 1.3 aren't applicable to the next version, 1.4. When you retrieve the file for editing, you can use the `-x` option to *exclude* delta 1.3 from the working copy:

```
hermes% sccs edit -x1.3 program.c
```

Now, when you check in delta 1.5, that delta will include the changes made in delta 1.4, but not those from delta 1.3. In fact, you can exclude a list of deltas by supplying a comma-separated list to `-x`, or a range of deltas, separated with a dash. For example, if you want to exclude 1.3 and 1.4, you could use:

```
hermes% sccs edit -x1.3,1.4 program.c
```

or

```
hermes% sccs edit -x1.3-1.4 program.c
```

In this example:

```
hermes% sccs edit -x1.3-1 program.c
```

SCCS excludes the range of deltas from 1.3 to the current highest delta in release 1.

In certain cases when using –x there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS displays a message telling the range of lines affected. Examine these lines carefully to see if the version SCCS derived is correct.

Since each delta (in the sense of "a set of changes") can be excluded at will, it is most useful to include a related set of changes within each delta.

Combining Versions: `sccs comb`

The `comb` subcommand generates a Bourne Shell script that, when run, constructs a new history file in which selected deltas are combined or eliminated. This can be useful when disk space is at a premium.

CAUTION **In combining several deltas, the `comb`-generated script destroys a portion of the file's version log, including comments.**

The `-p`*sid* option indicates the oldest delta to preserve in the reconstruction. Another option,

`-c` *sid-list*

allows you to specify a list of deltas to include. *sid-list* is a comma-separated list; you can specify a range between two SIDs by separating them with a dash ('-') in the list. `-p` and `-c` are exclusive. The `-o` option attempts to minimize the number of deltas in the reconstruction.

The `-s` option produces a script that compares the size of the reconstruction with that of the original. The comparision is given as a percentage of the original the reconstruction would occupy, based on the number of blocks in each.

NOTE *When using comb, it is a good idea to keep a copy of the original history file on hand. While* `comb` *is intended to save disk space, it may not always. In some cases, it is possible that the resulting history file may be larger than the original.*

If no options are specified, `comb` preserves the minimum number of ancestors needed to preserve the changes made so far.

4.3. Version Control for Binary Files

Although SCCS is typically used for source files containing ASCII text, the SunOS version of SCCS allows you to apply version control to *binary* files as well (files that contain NULL or control characters, or do not end with a NEWLINE). The binary files are encoded[3] into an ASCII representation when checked in; working copies are decoded when retrieved.

You can use SCCS to track changes to files such as icons, raster images, and screen fonts.

You can use `sccs create -b` to force SCCS to treat a file as a binary file. When you `create` or `delta` a binary file, you get the warning message:

```
Not a text file (ad31)
```

You may also get the message:

```
No id keywords (cm7)
```

These messages may safely be ignored. Otherwise, everything proceeds as expected:

```
hermes% sccs create special.font

special.font:
Not a text file (ad31)
No id keywords (cm7)
1.1
20 lines
No id keywords (cm7)
hermes% sccs get special.font
1.1
20 lines
hermes% file special.font SCCS/s.special.font
special.font:   vfont definition
SCCS/s.special.font:    sccs
```

Use SCCS to control the updates to source files, and `make` to compile objects consistently.

Since the encoded representation of a binary file can vary significantly between versions, history files for binary sources can grow at a much faster rate than those for ASCII sources. However, using the same version control system for all source files makes dealing with them much easier.

[3] See `uuencode`(1C) for details.

4.4. Maintaining Source Directories

When using SCCS, it is the history files, and not the working copies, that are the real source files.

Duplicate Source Directori

If you are working on a project and wish to create a duplicate set of sources for some private testing or debugging, you can make a symbolic link to the `SCCS` subdirectory in your private working directory:

```
hermes% cd /private/working/cmd.dir
hermes% ln -s /usr/src/cmd/SCCS SCCS
```

This makes it a simple matter to retrieve a private (duplicate) set of working copies, of the source files using:

```
sccs get SCCS
```

While working in the duplicate directory, you can also check files in and out—just as you could if you were in the original directory.

SCCS and `make`

SCCS is often used with `make`(1) to maintain a software project. The SunOS version of `make` provides for automatic retrieval of source files. (Other versions of `make` provide special rules that accomplish the same purpose.) It is also possible to retrieve earlier versions of all the source files, and to use `make` to rebuild earlier versions of the project:

```
hermes% mkdir old.release ; cd old.release
hermes% ln -s ../SCCS SCCS
hermes% sccs get -c"87/10/01" SCCS
SCCS/s.Makefile:
1.3
47 lines
...
hermes% make
...
```

As a general rule, no one should check in source files while a build is in progress. When a project is about to be released, all files should be checked in before it is built. This insures that the sources for a released project are stable.

Keeping SIDs Consistent Across Files

With some care, it is possible to keep the SIDs consistent across sources composed of multiple files. The trick here is to `edit` all the files at once. The changes can then be made to whatever files are necessary; check in all the files (even those not changed). This can be done fairly easily by specifying the `SCCS` subdirectory as the filename argument to both `edit` and `delta`:

```
hermes% sccs edit SCCS
...
hermes% sccs delta SCCS
```

With the `delta` subcommand, you are prompted for comments only once; the comment is applied to all files being checked in. To determine which files have

changed, you can compare the "lines added, deleted, unchanged" fields in each file's delta table.

Starting a New Release

To create a new release of a program, specify the release number you want to create when you check the file out for editing, using the `-r`*n* option to `edit`; *n* is the new release number:

```
hermes% sccs edit -r2 program.c
```

In this case, when the new version is `delta`'ed, it will be the first level delta in release 2, with SID 2.1. To change the release number for all SCCS-files in the directory, use:

```
hermes% sccs edit -r2 SCCS
```

Temporary Files used by SCCS

When SCCS modifies an `s.` file (that is, a history file), it does so by writing to a temporary copy called an `x.` file. When the update is complete, SCCS uses the `x.` file to overwrite the old `s.` file. This insures that the history file is not damaged when processing terminates abnormally. The `x.` file is created in the same directory as the history file, is given the same permissions, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, subcommands that update the history create a lock file, called a `z.` file, which contains the PID of the process performing the update. Once the update has completed, the `z.` file is removed. The `z.` file is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user.

4.5. Branches

You can think of the deltas applied to an SCCS file as the nodes of a tree; the root is the initial version of the file. The root delta (node) is number '1.1' by default, and successor deltas (nodes) are named '1.2', '1.3', and so forth. As noted earlier, these first two parts of the SID are the release and level numbers. The naming of a successor to a delta proceeds by incrementing the level number. You have also seen how to check out a new release when a major change to the file is made. The new release number applies to all successor deltas as well, unless you specify a new level in a prior release.

Thus, the evolution of a particular file may be represented as follows:

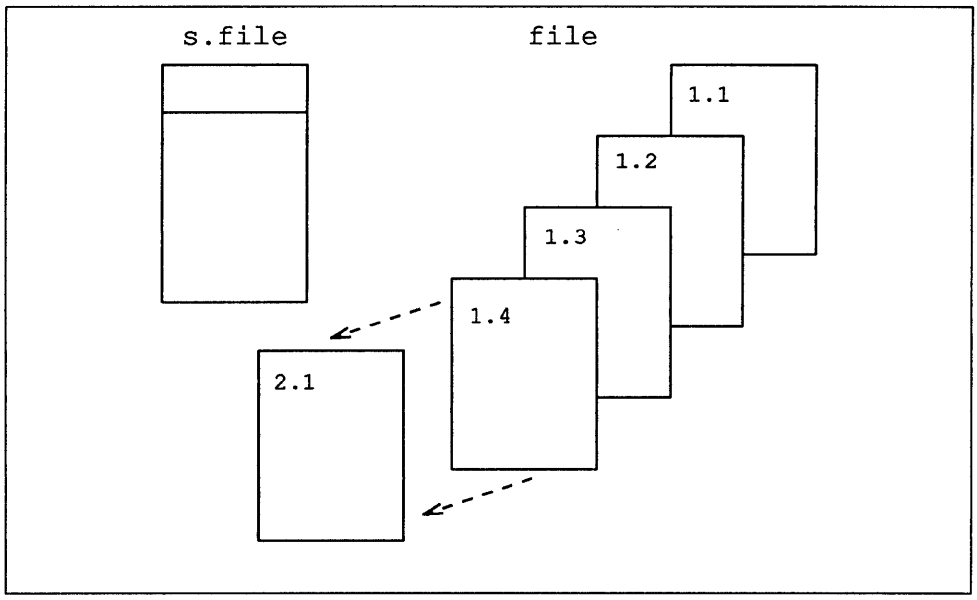


Figure 4-1 *Evolution of an SCCS File*

We can call this structure the 'trunk' of the SCCS delta tree. It represents the normal sequential development of an SCCS file; changes that are part of any given delta depend upon all the preceding deltas.

However, situations can arise when it is convenient to create an alternate branch on the tree. For instance, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas. Assume that a user reports a problem in version 1.3 which cannot wait until release 2 to be corrected. The changes necessary to correct the problem will have to be applied as a delta to version 1.3. This requires the creation of a new version, but one that is independent of the work being done for release 2. The new delta will thus occupy a node on a new branch of the tree.

The SID for a branch delta consists of four parts: the release and level numbers, and the *branch* and *sequence* numbers:

release . level . branch . sequence

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta; the first such branch is 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.1 identifies the first delta of the first branch derived from delta 1.3, as shown below.

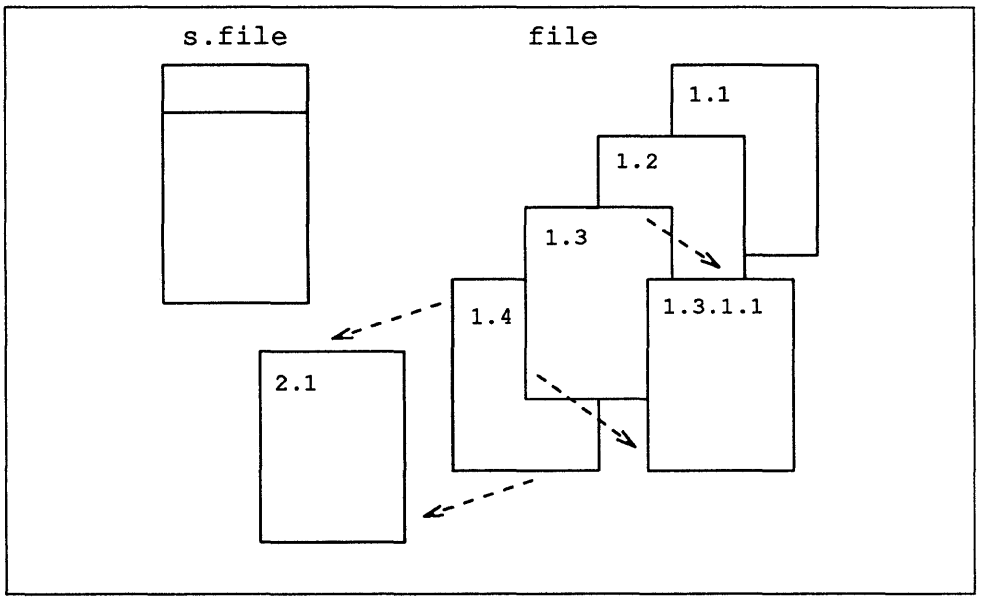


Figure 4-2 *Tree Structure with Branch Deltas*

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

The first two components of the name of a branch delta are always those of the ancestral trunk delta. The branch component is assigned in the order of creation on the branch, independent of its location relative to the trunk. Thus, a branch delta may always be identified as such from its name, and while the trunk delta may be identified from the branch delta's name, it is *not* possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named '1.3.1.*n*'. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named '1.3.2.*n*'. The only information that may be derived from the name of delta 1.3.2.2 is that it is the second chronological delta on the second chronological branch whose trunk ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

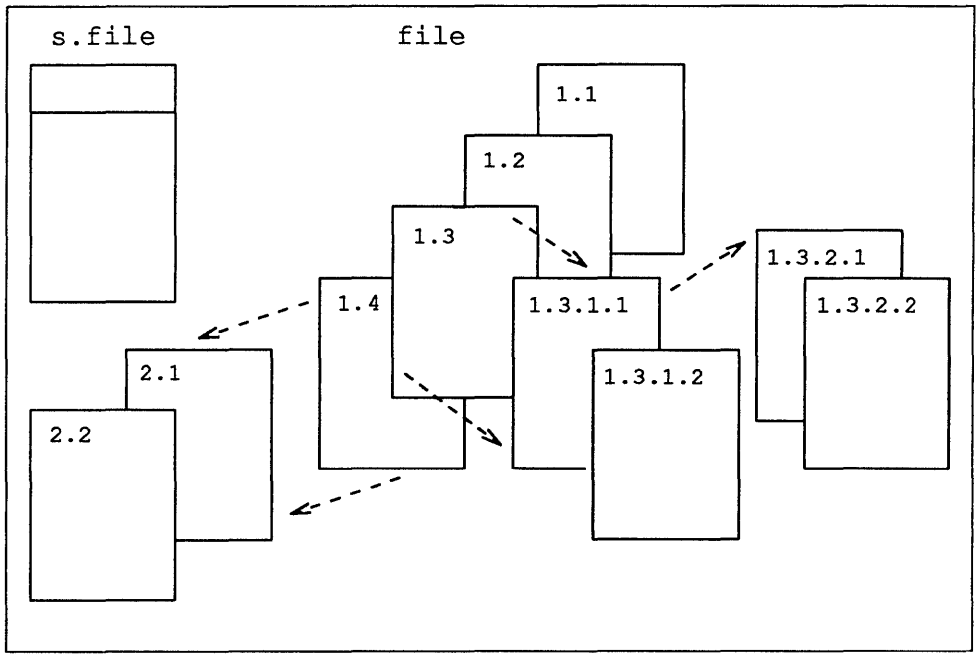


Figure 4-3 *Extending the Branching Concept*

Branch deltas allow the generation of arbitrarily complex tree structures. It is best to keep the use of branches to a minimum.

Using Branches

You can use branches when you need to keep track of an alternate versions developed in parallel, such as for bug fixes or experimental purposes.

Before you can create a branch, you must enable the "branch" flag in the history file using the `sccs admin` command, as follows:

```
hermes% sccs admin -fb program.c
```

The `-fb` option sets the `b` (branch) flag in the history file.

Creating a Branch Delta

To create a branch from delta 1.3, for `program.c` you would use the `sccs edit` subcommand shown below:

```
hermes% sccs edit -r1.3 -b program.c
```

When you check in your edited version, the branch delta will have SID 1.3.1.1. Subsequent deltas made from this branch will be numbered 1.3.1.2, and so on.

Retrieving Versions From Branch Deltas

Branch deltas usually aren't included in the version retrieved by `get`. To retrieve a branch version (the version associated with a branch delta), you must specifically request it with the `-r` option. If you omit the sequence number, as in the next example, SCCS retrieves the highest delta in the branch:

```
hermes% sccs get -r1.3.1 program.c
1.3.1.1
87 lines
```

4.6. Administering SCCS Files

By convention, history files and all temporary SCCS files reside in the `SCCS` subdirectory. In addition to the standard file protection mechanisms, SCCS allows certain releases to be frozen, and access to releases to be restricted to certain users (see `sccs-admin`(1) for details). History files normally have permissions set to 444 (read-only for everyone), to prevent modification by utilities other than SCCS. In general, it is not a good idea to edit the history files.

A history file should have just one link. SCCS utilities update the history file by writing out a modified copy (`x.file`), and then renaming the copy.

Interpreting Error Messages: `sccs help`

The `help` subcommand displays information about SCCS error messages and utilities.

`help` normally expects either the name of an SCCS utility, or the code (in parentheses) from an SCCS error message. If you supply no argument, `help` prompts for one. The directory `/usr/lib/help` contains files with the text of the various messages `help` displays.

Altering History File Defaults: `sccs admin`

There are a number of parameters that can be set using the *admin* command. The most interesting of these are flags. Flags can be added by using the `-f` option. For example:

```
hermes% sccs admin -fdl program.c
```

sets the 'd' flag to the value '1'. This flag can be deleted by using:

```
hermes% sccs admin -dd program.c
```

The most useful flags are:

`b` Allow branches to be made using the `-b` option to `sccs edit` (see *Branches*, above).

`d`*SID*
Default SID to be used on an `sccs get` or `sccs edit`. If this is just a release number it constrains the version to a particular release only.

`i` Give a fatal error if there are no ID keywords in a file. This prevents a version from being checked in when the ID keywords are missing or expanded by mistake.

`y` The value of this flag replaces the `%Y%` ID keyword.

`-t`*file*
store descriptive text from *file* in the `s.` file. This descriptive text might be the documentation or a design and implementation document. Using the `-t` option ensures that if the `s.` file is passed on to someone else, the

documentation will go along with it. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use `prt -t`.

The `sccs admin` command can be used safely any number of times on files. A current version need not be retrieved for `admin` to work.

Validating the History File

You can use the `val` subcommand to check certain assertions about a history file. `val` always checks for the following conditions:

- A corrupted history file.
- The history file can't be opened for reading, or the file is not an SCCS history.

If you use the `-r` option, `val` checks to see if the indicated SID exists.

Restoring the History File

In particularly bad circumstances, the history file itself may get corrupted. The most common way this happens is for someone to edit it. Since the file contains a checksum, you will get errors every time you read a corrupted file. To correct the checksum, use:

```
hermes% sccs admin -z program.c
```

CAUTION **When SCCS says that the history file is corrupted, it may indicate serious damage beyond an incorrect checksum. Be careful to safeguard your current changes before attempting to correct a history file.**

4.7. Reference Tables

Table 4-1 *SCCS ID Keywords*

Keyword	Expands to
`%Z%`	`@(#)` (search string for the `what` command)
`%M%`	The current module (file) name
`%I%`	The highest SID applied
`%W%`	shorthand for: `%Z%%M%` *tab* `%I%`
`%G%`	The date of the delta corresponding to the `%I%` keyword.
`%R%`	The current *release* number.
`%Y%`	The value of the `t` flag (set by `sccs admin`).

Table 4-2 *SCCS Utility Commands*

SCCS Utility Programs	
Command	*Refer to:*
`admin`	`sccs-admin(1)`
`cdc`	`sccs-cdc(1)`
`comb`	`sccs-comb(1)`
`delta`	`sccs-delta(1)`
`get`	`sccs-get(1)`
`help`	`sccs-help(1)`
`prs`	`sccs-prs(1)`
`rmdel`	`sccs-rmdel(1)`
`sact`	`sccs-sact(1)`
`sccsdiff`	`sccs-sccsdiff(1)`
`unget`	`sccs-unget(1)`
`val`	`sccs-val(1)`
`what`*	`what(1)`

*`what` is a general-purpose command.

Table 4-3 *Data Keywords for* `prs -d`

Keyword	***Data Item***	***File Section***	***Value***	***Format***†
`:Dt:`	Delta information	Delta Table	see below*	S
`:DL:`	Delta line statistics	"	`:Li:/:Ld:/:Lu:`	S
`:Li:`	Lines inserted by Delta	"	*nnnnn*	S
`:Ld:`	Lines deleted by Delta	"	*nnnnn*	S
`:Lu:`	Lines unchanged by Delta	"	*nnnnn*	S
`:DT:`	Delta type	"	`D` *or* `R`	S
`:I:`	SCCS ID string (SID)	"	`:R:.:L:.:B:.:S:`	S
`:R:`	Release number	"	*nnnn*	S
`:L:`	Level number	"	*nnnn*	S
`:B:`	Branch number	"	*nnnn*	S
`:S:`	Sequence number	"	*nnnn*	S
`:D:`	Date Delta created	"	`:Dy:/:Dm:/:Dd:`	S
`:Dy:`	Year Delta created	"	*nn*	S
`:Dm:`	Month Delta created	"	*nn*	S
`:Dd:`	Day Delta created	"	*nn*	S
`:T:`	Time Delta created	"	`:Th:.:Tm:.:Ts:`	S
`:Th:`	Hour Delta created	"	*nn*	S
`:Tm:`	Minutes Delta created	"	*nn*	S
`:Ts:`	Seconds Delta created	"	*nn*	S
`:P:`	Programmer who created Delta	"	*logname*	S
`:DS:`	Delta sequence number	"	*nnnn*	S
`:DP:`	Predecessor Delta seq-no.	"	*nnnn*	S
`:DI:`	Sequence number of deltas included, excluded, ignored	"	`:Dn:/:Dx:/:Dg:`	S

Table 4-3 *Data Keywords for* `prs -d`*— Continued*

Keyword	*Data Item*	*File Section*	*Value*	*Format*†
`:Dn:`	Deltas included (seq #)	"	`:DS: :DS:...`	S
`:Dx:`	Deltas excluded (seq #)	"	`:DS: :DS:...`	S
`:Dg:`	Deltas ignored (seq #)	"	`:DS: :DS:...`	S
`:MR:`	MR numbers for delta	"	*text*	M
`:C:`	Comments for delta	"	*text*	M
`:UN:`	User names	User Names	*text*	M
`:FL:`	Flag list	Flags	*text*	M
`:Y:`	Module type flag	"	*text*	S
`:MF:`	MR validation flag	"	yes *or* no	S
`:MP:`	MR validation pgm name	"	*text*	S
`:KF:`	Keyword error/warning flag	"	yes *or* no	S
`:BF:`	Branch flag	"	yes *or* no	S
`:J:`	Joint edit flag	"	yes *or* no	S
`:LK:`	Locked releases	"	`:R:...`	S
`:Q:`	User defined keyword	"	*text*	S
`:M:`	Module name	"	*text*	S
`:FB:`	Floor boundary	"	`:R:`	S
`:CB:`	Ceiling boundary	"	`:R:`	S
`:Ds:`	Default SID	"	`:I:`	S
`:ND:`	Null delta flag	"	yes *or* no	S
`:FD:`	File descriptive text	Comments	*text*	M
`:BD:`	Body	Body	*text*	M
`:GB:`	Gotten body	"	*text*	M
`:W:`	A form of *what*(1) string	N/A	`:Z::M:\t:I:`	S
`:A:`	A form of *what*(1) string	N/A	`:Z::Y: :M: :I::Z:`	S
`:Z:`	*what*(1) string delimiter	N/A	`@(#)`	S
`:F:`	SCCS file name	N/A	*text*	S
`:PN:`	SCCS file path name	N/A	*text*	S

†S = single-line format, M = multi-line

* `:Dt: = :DT: :I: :D: :T: :P: :DS: :DP:`

5

make User's Guide

5.1. Overview

This chapter describes Sun's version of the make utility, which includes important features such as hidden dependency checking, command dependency checking, pattern-matching rules, and automatic retrieval of SCCS files. This version can run successfully with makefiles written for previous versions of make. However, makefiles that rely on Sun's enhancements may not be compatible with other versions of this utility. Refer to Appendix A, make *Enhancements Summary* for a complete summary of Sun's enhancements and compatibility issues.

make streamlines the process of generating and maintaining object files and executable programs. It helps you to compile programs consistently, and eliminates unnecessary recompilation of modules that are unaffected by source code changes.

make provides a number of features that simplify compilations, but you can also use it to automate any complicated or repetitive task that isn't interactive. You can use make to update and maintain object libraries, to run test suites, and to install files onto a filesystem or tape. In conjunction with SCCS, you can use make to insure that a large software project is built from the desired versions in an entire hierarchy of source files.

make reads a file that you create, called a *makefile*, which contains information about what files to build and how to build them. Once you write and test the makefile, you can forget about the processing details; make takes care of them. This gives you more time to concentrate on improving your code; the repetitive portion of the maintenance cycle is reduced to:

think — edit — **make** — test . . .

Dependency Checking: make **vs. Shell Scripts**

While it is possible to use a shell script to assure consistency in trivial cases, scripts to build software projects are often inadequate. On the one hand, you don't want to wait for a simple-minded script to compile every single program or object module when only one of them has changed. On the other hand, having to edit the script for each iteration can defeat the goal of consistency. Although it is possible to write a script of sufficient complexity to recompile only those modules that require it, make does this job better.

make allows you to write a simple, structured listing of what to build and how to build it. It uses the mechanism of *dependency checking* to compare each module with the source or intermediate files it derives from. make only rebuilds a module if one or more of these prerequisite files, called *dependency files*, has changed since the module was last built. To determine whether a derived file is out of date with respect to its sources, make compares the modification time of the (existing) module with that of its dependency file. If the module is missing, or if it is older than the dependency file, make considers it to be out of date, and issues the commands necessary to rebuild it.

Optionally, a module can be treated as out of date if the commands used to build it have changed.

Because `make` does a complete dependency scan, changes to a source file are consistently propagated through any number of intermediate files or processing steps. This lets you specify a hierarchy of steps in a top-down fashion.

You can think of a makefile as a recipe. `make` reads the recipe, decides which steps need to be performed, and executes only those steps that are required to produce the finished module. Each file to build, or step to perform, is called a *target*. The makefile entry for a target contains its name, a list of targets on which it depends, and a list of commands for building it. The list of commands is called a *rule*. `make` treats dependencies as prerequisite targets, and updates them (if necessary) before processing its current target. The rule for a target need not always produce a file, but if it does, the file for which the target is named is referred to as the *target file*. Each file from which a target is derived (e.g., that the target depends on) is called a *dependency file*.

If the rule for a target produces no file by that name, `make` performs the rule and considers the target to be up-to-date for the remainder of the run.

`make` assumes that only *it* will make changes to files being processed during the current run. If a source file is changed by another process while `make` is running, the files it produces may be in an inconsistent state.

Writing a Simple Makefile

The basic format for a makefile target entry is:

Figure 5-1 *Makefile Target Entry Format*

If there is no rule for a target entry, `make` looks for an implicit rule to use.

```
target ... : [ dependency ... ]
             [ command ]
             ...
```

If the dependency list is terminated with a semicolon and followed by a command, that command is included in the rule. However, makefiles tend to read better if you avoid this.

In the first line, the list of target names is terminated by a colon. This, in turn, is followed by the dependency list if there is one. If several targets are listed, this indicates that each such target is to be built independently using the rule supplied.

Subsequent lines that start with a TAB are taken as the commands lines that comprise the target's rule. A common error is to use SPACE characters instead of the leading TAB.

Lines that start with a # are treated as comments up until the next (unescaped) NEWLINE, and do not terminate the target entry. The target entry is terminated by the next nonempty line that begins with a character other than TAB or #, or by the end of the file.

A trivial makefile might consist of just one target:

Figure 5-2 *A Trivial Makefile*

```
test:
        ls test
        touch test
```

When you run `make` with no arguments, it searches first for a file named `makefile`, or if there is no file by that name, `Makefile`. If either of these files is under SCCS control, `make` checks the makefile against its history file. If it is out of date, `make` extracts the latest version.

The convention is to use the name `Makefile`, since filenames starting with a capital are listed first by `ls`; this highlights the fact that a makefile is present.

If `make` finds a makefile, it begins the dependency check with the first target entry in that file. Otherwise you must list the targets to build as arguments on the command line. `make` displays each command it runs while building its targets.

```
hermes% make
ls test
test not found
touch test
hermes% ls test
test
```

Because the file `test` was not present (and therefore out of date), `make` performed the rule in its target entry. If you run `make` a second time, it issues a message indicating that the target is now up to date:

```
hermes% make
'test' is up to date.
```

and skips the rule.

Line breaks within a rule are significant in that each command line is performed by a separate process or shell.

`make` invokes a Bourne shell to process a command line if that line contains any shell metacharacters, such as a semicolon (`;`), redirection symbols (<, >, >>, |), substitution symbols (`*`, `?`, `[ ]`, `$`, =), or quotes, escapes or comments (", ', `, \, #, etc.:), If a shell isn't required to parse the command line, *make* `exec()`'s the command directly.

This means that a rule such as:

```
test:
        cd /tmp
        pwd
```

behaves differently than you might expect, as shown below.

```
hermes% make test
cd /tmp
pwd
/usr/tutorial/waite/arcana/minor/pentangles
```

You can use semicolons to specify a sequence of commands to perform in a single shell invocation:

```
test:
        cd /tmp ; pwd
```

Or, you can continue the input line onto the next line in the makefile by escaping the NEWLINE with a backslash (\). The escaped NEWLINE is treated as white space by make.

The backslash must be the last character on the line. The semicolon is required by the shell.

```
test:
        cd /tmp ; \
          pwd
```

Basic Use of Implicit Rules

When there is no rule given for a specified target, make attempts to use an *implicit rule* to build it. When make finds a rule for the class of files the target belongs to, it applies the rule listed in the implicit rule's target entry.

In addition to any makefile(s) that you supply, make reads in the default makefile, /usr/include/make/default.mk, which contains the target entries for a number of implicit rules, along with other information.[4]

There are two types of implicit rules. *Suffix* rules specify a set of commands for building a file with one suffix from another file with the same basename but a different suffix. *Pattern-matching* rules select a rule based on a target and dependency that match respective wild-card patterns. The implicit rules provided by default are suffix rules.

In some cases, the use of suffix rules can eliminate the need for writing a makefile entirely. For instance, to build an object file named functions.o from a single C source file named functions.c, you could use the command:

```
hermes% make functions.o
cc   -sun4 -c functions.c -o functions.o
```

This would work equally well for building the object file nonesuch.o from the source file nonesuch.c.

[4] Implicit rules were hard-coded in earlier versions of make.

To build an executable file named `functions` (with a null suffix) from `functions.c`, you need only type the command:

```
hermes% make functions
cc   -sun4  -o functions functions.c
```

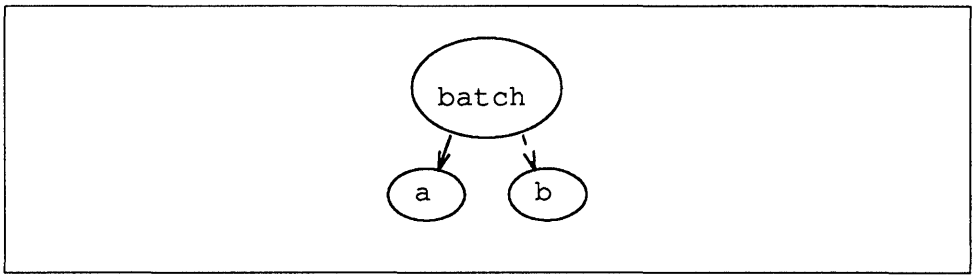
The rule for building a `.o` file from a `.c` file is called the `.c.o` (pronounced "dot-see-dot-oh") suffix rule. The rule for building an executable program from a `.c` file is called the `.c` ("dot-see") rule. The complete set of default suffix rules is listed in Table 5-1 .

Processing Dependencies

Once `make` begins, it processes targets as it encounters them in its depth-first dependency scan. For example, with the following makefile:

```
batch: a b
        touch batch
b:
        touch b
a:
        touch a
c:
        echo "you won't see me"
```

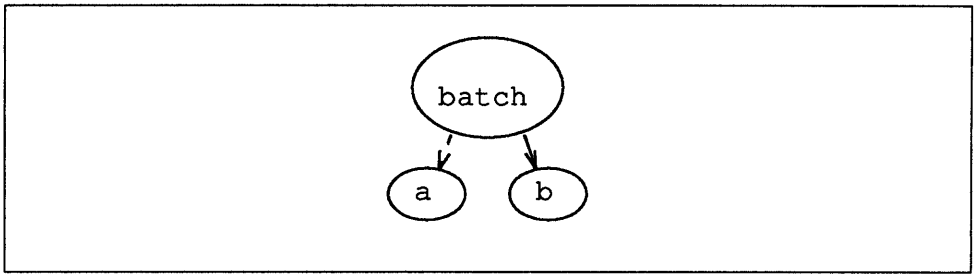
`make` starts with the target `batch`. Since `batch` has some dependencies that haven't been checked yet, namely `a` and `b`, `make` defers `batch` until after it has checked them against any dependencies they might have.



Since `a` has no dependencies, `make` processes it; if the file is not present `make` performs the rule in its target entry.

```
hermes% make
touch a
...
```

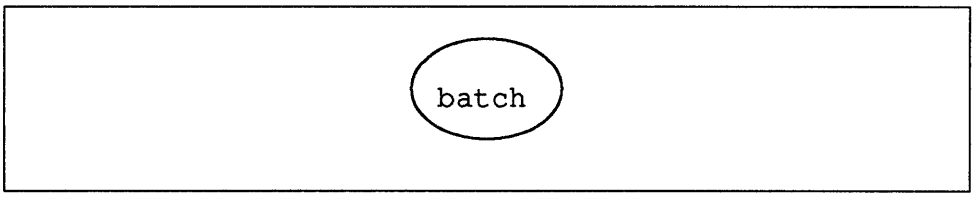
Next, `make` works its way back up to the parent target `batch`. Since there is still an unchecked dependency `b`, `make` descends to `b` and checks it.



b also has no dependencies, so make performs its rule:

```
. . .
touch b
. . .
```

Finally, now that all of the dependencies for batch have been checked and built (if needed), make checks batch.



Since it rebuilt at least one of the dependencies for batch, make assumes that batch is out of date and rebuilds it; if a or b had not been built in the current make run, but were present in the directory and newer than batch, make's timestamp comparison would also result in batch being rebuilt:

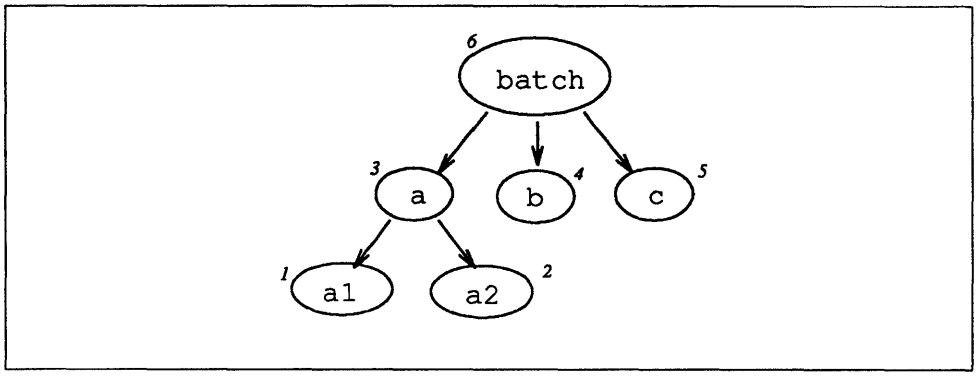
```
. . .
touch batch
```

Target entries that aren't encountered in a dependency scan are not processed. Although there is a target entry for c in the makefile, make does not encounter it while performing the dependency scan for batch, so its rule is not performed. Target entries that aren't encountered in a dependency scan are not processed. You can select an alternate starting target like c by entering it as an argument to the make command.

In the next example, the batch target produces no file. Instead, it is used as a label to group a set of targets.

```
batch:  a b c
a:   a1  a2
        touch a
b:
        touch b
c:
        touch c
a1:
        touch a1
a2:
        touch a2
```

In this case, the targets are checked and processed as shown in the following diagram:



- make checks batch for dependencies and notes that there are three, and so defers it.
- make checks a, the first dependency, and notes that it has two dependencies of its own. So, continuing in like fashion, make:

1. Checks a1, and if necessary, rebuilds it.
2. Checks a2, and if necessary, rebuilds it.
3. Determines whether to build a.
4. Checks b and rebuilds it if need be.
5. Checks and rebuilds c if needed.
6. After traversing its dependency tree, make checks and processes the topmost target, batch. If batch contained a rule, make would perform that rule. Since batch has no rule, make performs no action, but notes that batch has been rebuilt; any targets depending on batch would also be rebuilt.

Null Rules

If a target entry contains no rule, `make` attempts to select an implicit rule to build it. If `make` cannot find an appropriate implicit rule and there is no SCCS history from which to retrieve it, `make` concludes that the target has no corresponding file, and regards the missing rule as a null rule. With this makefile:

You can use a dependency with a null rule to force the target's rule to be executed. The conventional name for such a dependency is `FORCE`.

```
haste: FORCE
        echo "haste makes waste"
FORCE:
```

`make` performs the rule for making `haste`, even if a file by that name is up to date:

```
hermes% touch haste
hermes% make haste
echo "haste makes waste"
haste makes waste
```

Unknown Targets

If a target is named either on the command line or in a dependency list, and it:

- is not a file present in the working directory,
- has no target or dependency entry,
- does not belong to a class of files for which an implicit rule is defined, and
- has no SCCS history file,
- there is no rule specified for the `.DEFAULT` special target

`make` stops processing and issues an error message.[5]

```
hermes% make believe
make: Fatal error: Don't know how to make target 'believe'.
```

Running Commands Silently

You can inhibit the display of a command line within a rule by inserting an `@` as the first non-TAB character on that line. For example, the following target:

```
quiet:
        @ echo you only see me once
```

produces:

```
hermes% make quiet
you only see me once
```

[5] However, if the `-k` option is in effect, `make` will continue with other targets that do not depend on the one in which the error occurred.

Special-function targets begin with a dot (.). Target names that begin with a dot are never used as the starting target, unless specifically requested as an argument on the command line.

If you want to inhibit the display of commands during a particular make run, you can use the -s option. If you want to inhibit the display of all command lines in every run, add the special target .SILENT

to your makefile:

```
.SILENT:
quiet:
        echo you only see me once
```

Ignoring a Command's Exit Status

make normally issues an error message and stops when a command returns a nonzero exit code. For example, if you have the target:

```
rmxyz:
        rm xyz
```

and there is no file named xyz, make halts after rm returns its exit status.

```
hermes% ls xyz
xyz not found
hermes% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target `rmxyz'
```

If - and @ are the first two such characters, both take effect.

To continue processing regardless of the command's exit code, use a dash character (-) as the first non-TAB character:

```
rmxyz:
        -rm xyz
```

In this case you get a warning message indicating the exit code make received:

```
hermes% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

Unless you are testing a makefile, it is usually a bad idea to ignore non-zero error codes on a global basis.

Although it is generally ill-advised to do so, you can have make ignore error codes entirely with the -i option. You can also have make ignore exit codes when processing a given makefile, by including the .IGNORE special target, though this too should be avoided.

If you are processing a list of targets, and you want make to continue with the next target on the list rather than stopping entirely after encountering a non-zero

return code, use the –k option.

Automatic Retrieval of SCCS Files

When source files are named in the dependency list, `make` treats them just like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, `make` assumes that that file is up to date. If, however, a source file is under SCCS control, `make` does some additional checking to assure that the source file is up to date. If the file is missing, or if the history file is newer, `make` automatically issues an

```
sccs get -s filename -Gfilename
```

command to retrieve the most recent version:[6] However, if the source file is writable by anyone, `make` does not retrieve a new version.

```
hermes% ls SCCS/*
SCCS/s.functions.c
hermes% rm -f functions.c
hermes% make functions
sccs get -s functions.c -Gfunctions.c
cc -sun4 -o functions functions.c
```

`make` only checks the timestamp of the retrieved version against the timestamp of the history file. It does *not* check to see if the version present in the directory is the most recently checked-in version. So, if someone has done a get by date (`sccs get -c`), `make` would not discover this fact, and you might unwittingly build an older version of the program or object file. To be absolutely sure that you are compiling the latest version, you can precede `make` with an "`sccs get SCCS`" or an "`sccs clean`" command.

Suppressing SCCS Retrieval

The command for retrieving SCCS files is specified in the rule for the `.SCCS_GET` special target in the default makefile. To suppress automatic retrieval, simply add an entry for this target with an empty rule to your makefile:

```
#   Suppress sccs retrieval.
.SCCS_GET:
```

Passing Parameters: Simple `make` Macros

`make`'s macro substitution comes in handy when you want to pass parameters to commands lines within a makefile. Suppose that you sometimes wish to compile an optimized version of the program `program` using `cc`'s `-O` option. You can lend this sort of flexibility to your makefile by adding a *macro reference*, such as the one below, to the target for `functions`:

[6] With other versions of `make` automatic SCCS retrieval was a feature only of certain implicit rules. Also, unlike earlier versions, `make` only looks for history (`s.`) files in the SCCS subdirectory; history files in the current directory are ignored.

```
functions: functions.c
        cc -sun4 $(CFLAGS) -o functions functions.c
```

The macro reference acts as a placeholder for a value that you define, either in the makefile itself, or as an argument to the `make` command. If you then supply `make` with a *definition* for the `CFLAGS` macro, `make` replaces its references with the value you have defined.

There is a reference to the `CFLAGS` macro in both the `.c` and the `.c.o` implicit rules.

The command-line definition must be a single argument, hence the quotes in this example.

```
hermes% rm functions
hermes% make functions "CFLAGS= -O"
cc -sun4 -O -o functions functions.c
```

If a macro is undefined, `make` expands its references to an empty string.

You can also include macro definitions in the makefile itself. A typical use is to set `CFLAGS` to `-O`, so that `make` produces optimized object code by default:

```
CFLAGS= -O
functions: functions.c
        cc -sun4 $(CFLAGS) -o functions functions.c
```

A macro definition supplied as a command line argument to `make` overrides other definitions in the makefile.[7] For instance, to compile `functions` for debugging with `dbx` or `dbxtool`, you can define the value of `CFLAGS` to be `-g` on the command line:

```
hermes% rm functions
hermes% make CFLAGS=-g
cc -sun4 -g -o functions functions.c
```

To compile a profiling variant for use with `gprof`, supply both `-O` and `-pg` in the value for `CFLAGS`.

A macro reference must include parentheses when the name of the macro is longer than one character. If the macro name is only one character, the parentheses can be omitted. You can use curly braces, `{` and `}`, instead of parentheses. For example, '`$X`', '`$(X)`', and '`${X}`' are equivalent.

Command Dependency Checking and `.KEEP_STATE`

In addition to the normal dependency checking, you can use the special target `.KEEP_STATE` to activate *command dependency* checking.[8] When activated, `make` not only checks each target file against its dependency files, it compares each command line in the rule with those it ran the last time the target was built. This information is stored in a state file in the working directory.

[7] Conditionally defined macros are an exception to this. Refer to *Conditional Macro Definitions* for details.

[8] This feature is not available in earlier versions of `make`.

With the makefile:

```
CFLAGS= -O
.KEEP_STATE:

functions: functions.c
        cc -sun4 -o functions functions.c
```

the following commands work as shown:

```
hermes% make
cc -sun4 -O -o functions functions.c
hermes% make CFLAGS=-g
cc -sun4 -g -o functions functions.c
hermes% make "CFLAGS= -O -pg"
cc -sun4 -O -pg -o functions functions.c
```

This assures you that `make` compiles a program with the options you want, even if a different variant is present and otherwise up to date.

The first `make` run with `.KEEP_STATE` in effect recompiles all targets.

The `KEEP_STATE` variable, when imported from the environment, has the same effect as the `.KEEP_STATE` target.

Suppressing or Forcing Command Dependency Checking for Selected Lines

To suppress command dependency checking for a given command line, insert a question mark as the first character after the TAB.

Command dependency checking is automatically suppressed for lines containing the dynamic macro `$?`. This macro stands for the list of dependencies that are newer than the current target, and can be expected to differ between any two `make` runs.[9] To force `make` to perform command dependency checking on a line containing this macro, prefix the command line with a `!` character (following the TAB).

The State File

When `.KEEP_STATE` is in effect, `make` writes out a state file named `.make.state`, in the current directory. This file lists all targets that have ever been processed while `.KEEP_STATE` has been in effect, along with the rules to build them, in makefile format. In order to assure that this state file is maintained consistently, once you have added `.KEEP_STATE` to a makefile, we recommend that you leave it in effect.[10]

---

[9] See *Implicit Rules and Dynamic Macros* for more information.

[10] Since this target is ignored in earlier versions of `make`, it does not introduce any compatibility problems. Other versions simply treat it as a superfluous target that no targets depend on, with an empty rule and no dependencies of its own. Since it starts with a dot, it is not used as the starting target.

Hidden Dependencies and .KEEP_STATE

When a C source file contains `#include` directives for interpolating headers, the target depends just as much on those headers as it does on the sources that include them. Because such headers may not be listed explicitly as sources in the compilation command line, they are called *hidden dependencies*. When `.KEEP_STATE` is in effect, `make` receives a report from the various compilers and compilation preprocessors indicating which hidden dependency files were interpolated for each target.[11] It adds this information to the dependency list in the state file. In subsequent runs, these additional dependencies are processed just like regular dependencies. This feature maintains the hidden dependency list for each target automatically; it insures that the dependency list for each target is always accurate and up to date. It also eliminates the need for the complicated schemes found in some earlier makefiles to generate complete dependency lists.

A slight inconvenience can arise the first time `make` processes a target with hidden dependencies, because there is as yet no record of them in the state file. If a header is missing, and `make` has no record of it, `make` won't know that it needs to retrieve it from SCCS before compiling the target. So, even though there is an SCCS history file, the current version won't be retrieved because it doesn't yet appear in a dependency list or the state file. So, when the C preprocessor attempts to interpolate the header, it won't find it; the compilation fails.

Supposing that an `#include` directive for interpolating the header `hidden.h` is added to `functions.c`, and that the file `hidden.h` is somehow removed before the subsequent `make` run. The results would be:

```
hermes% rm -f hidden.h
hermes% make functions
cc -sun4 -O -o functions functions.c
functions.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'functions'
...
```

A simple workaround might be to make sure that the new header is extant before you run `make`. Or, if the compilation should fail (and assuming the header is under SCCS), you could retrieve it from SCCS manually:

```
hermes% sccs get hidden.h
1.1
10 lines
hermes% make functions
cc -sun4 -O -o functions functions.c
```

In all future cases, should the header turn up missing, `make` will know to build or retrieve it for you, because it will be listed in the state file as a hidden dependency.

[11] Also unavailable with earlier versions of `make`.

Note that with hidden dependency checking, the `$?` macro includes the names of hidden dependency files. This may cause unexpected behavior in existing makefiles that rely on `$?`.

Hidden Dependencies and `.INIT`

The problem with both of these approaches is that the first `make` in the local directory may fail due to a random condition in some other (include) directory. This might entail forcing someone to monitor a (first) build. To avoid this, you can use the `.INIT` target to retrieve known hidden dependencies files from SCCS. `.INIT` is a special target that, along with its dependencies, is built at the start of the `make` run. To be sure that `hidden.h` is present, you could add the following line to your makefile;

```
.INIT: hidden.h
```

Displaying Information About a `make` Run

Running `make` with the `-n` option displays the commands `make` is to perform, without executing them. This comes in handy when verifying that the macros in a makefile are expanded as expected. With the following makefile:

There is an exception to this however. `make` executes any command line containing a reference to the `MAKE` macro (i.e., `$(MAKE)` or `${MAKE}`), regardless of `-n`. So, it would be a very bad idea to include a line like: "`$(MAKE) ; rm -f *`" in your makefile.

```
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
        $(LINK.c) -o functions main.o data.o
```

`make -n` displays:

```
hermes% make -n
cc -O -sun4 -c main.c
cc -O sun4 -c data.c
cc -O  -sun4 -o functions main.o data.o
```

`make` has some other options that you can use to keep abreast of what it's doing and why:

Setting an environment variable named `MAKEFLAGS` can lead to complications, since `make` adds its value to the list of options. To prevent puzzling surprises, avoid setting this variable.

`-d` Displays the criteria by which `make` determines that a target is be out-of-date. Unlike `-n`, it *does* process targets, as shown below. This options also displays the value imported from the environment (null by default) for the `MAKEFLAGS` macro, which is described in detail in a later section.

```
hermes% make -d
MAKEFLAGS value:
   Building main.o using suffix rule for .c.o because it is out of date relative to main.c
cc -O  -mc68020 -c main.c
   Building functions because it is out of date relative to main.o
   Building data.o using suffix rule for .c.o because it is out of date relative to data.c
cc -O  -mc68020 -c data.c
   Building functions because it is out of date relative to data.o
cc -O  -mc68020 -o functions main.o data.o
```

`-dd` This option displays all dependencies `make` checks, including any hidden dependencies, in vast detail.

`-D` Displays the text of the makefile as it is read.

`-DD` Displays the makefile and the default makefile, the state file, and hidden dependency reports for the current `make` run.

Several `-f` options indicate the concatenation of the named makefiles.

`-f` *makefile*
`make` uses the named *makefile* (instead of `makefile` or `Makefile`).

`-p` Displays the complete set of macro definitions and target entries.

`-P` Displays the complete dependency tree for each target encountered.

Due to its potentially troublesome side effects, we recommend against using the `-t` (touch) option for `make`.

There is an option that can be used to shortcut `make` processing, the `-t` option. When run with `-t`, `make` does not perform the rule for building a target. Instead it uses `touch` to alter the modification time for each target that it encounters in the dependency scan. It also updates the state file to reflect what it built. This often creates more problems than it supposedly solves, and so we recommend that you exercise extreme caution if you do use it. Note that if there is no file corresponding to a target entry `touch` creates it.

`clean` is the conventional name for a target that removes derived files. It is useful when you want to start a build from scratch.

The following is one example of how *not* to use `make -t`. Suppose you have a target named `clean` that performed housekeeping in the directory by removing target files produced by `make`:

```
clean:
        rm functions main.o data.o
```

If you give the nonsensical command:

```
hermes% make -t clean
touch clean
hermes% make clean
'clean' is up to date.
```

you then have to remove the file `clean` before your housekeeping target can work once again.

For a complete listing of all `make` options, refer to `make`(1) in the *SunOS Reference Manual.*

5.2. Compiling Programs with `make`

Compilation Strategies

In previous examples you have seen how to compile a simple C program from a single source file, using both explicit target entries and implicit rules. Most C programs, however, are compiled from several source files. Many include library routines, either from one of the standard system libraries or from a user-supplied library. Although it may be easier to recompile and link a single-source program using a single `cc` command, it is usually more convenient to compile programs with multiple sources in stages—first, by compiling each source file into a separate object (`.o`) file, and then by linking the object files to form an executable (`a.out`) file. This method requires more disk space, but subsequent (repetitive) recompilations need be performed only on those object files for which the sources have changed, which saves time.

A Simple Makefile

The makefile below is not all that elegant, but it does the job.

Figure 5-3 *Simple Makefile for Compiling C Sources: Everything Explicit*

```
#   Simple makefile for compiling a program from
#   two C source files.

.KEEP_STATE:

functions: main.o data.o
        cc -sun4 -O -o functions main.o data.o

main.o: main.c
        cc -sun4 -O -c main.c

data.o: data.c
        cc -sun4 -O -c data.c

clean:
        rm functions main.o data.o
```

In this example, `make` produces the object files `main.o` and `data.o`, and the executable file `functions`:

```
hermes% make
cc -sun4 -o functions main.o data.o
cc -sun4 -O -c main.c
cc -sun4 -O -c data.c
```

Using make's Predefined Macros

The next example performs exactly the same function, but demonstrates the use of make's predefined macros for the indicated compilation commands. Using predefined macros eliminates the need to edit makefiles when the underlying compilation environment changes. They also provide access to the CFLAGS macro (and other FLAGS macros) for supplying compiler options from the command line. Predefined macros are also used extensively within make's implicit rules. The predefined macros in the following makefile are listed below.[12] They are generally useful for compiling C programs.

Macro names that end in the string FLAGS are used to pass options to a related compiler-command macro. It is good practice to use these macros for consistency and portability. It is also good practice to note the desired default values for them in the makefile.

COMPILE.c — The cc command line; composed of the values of CC, CFLAGS, CPPFLAGS, and TARGET_ARCH, as follows, along with the -c option.

```
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -target $(TARGET_ARCH:-%=%) -c
```

The root of the macro name, COMPILE, is a convention used to indicate that the macro stands for a compilation command line (to generate an object, or .o file). The .c suffix is a mnemonic device to indicate that the command line applies to .c (C source) files.

The complete list of all predefined macros is shown in Table 1.2, below.

LINK.c — The basic cc command line to link object files, like COMPILE.c, but without the -c option and with a reference to the LDFLAGS macro:

```
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -target $(TARGET_ARCH:-%=%)
```

CC — The value cc. (You can redefine the value to be the pathname of an alternate C compiler.)

CFLAGS — Options for the cc command; none by default.

CPPFLAGS — Options for cpp; none by default.

LDFLAGS — Options for the link editor, ld; none by default.

TARGET_ARCH — The target-architecture argument to cc for use when cross-compiling. The default is set by make to the value returned by the arch command. This macro must be defined when using Sun's optional cross-compilers. Refer to *Cross-Compilation on the Sun Workstation* for details.

[12] Predefined macros are used more extensively than in earlier versions of make. Not all of the predefined macros shown here are available with earlier versions.

Figure 5-4 *Makefile for Compiling C Sources Using Predefined Macros*

```
#   Makefile for compiling two C sources

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
        $(LINK.c) -o functions main.o data.o

main.o: main.c
        $(COMPILE.c) main.c

data.o: data.c
        $(COMPILE.c) data.c

clean:
        rm functions main.o data.o
```

Using Implicit Rules to Simplify a Makefile: Suffix Rules

Since the command lines for compiling `main.o` and `data.o` from their respective `.c` files are now functionally equivalent to the `.c.o` suffix rule, their target entries are, in a sense, redundant; `make` performs the same compilation whether they appear in the makefile or not. This next version of the makefile eliminates them, relying on the `.c.o` rule to compile the individual object files.

Figure 5-5 *Makefile for Compiling C Sources Using Suffix Rules*

```
#   Makefile for a program from two C sources
#   using suffix rules.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
        $(LINK.c) -o functions main.o data.o

clean:
        rm functions main.o data.o
```

A complete list of suffix rules appears in Table 3-1.

As `make` processes the dependencies `main.o` and `data.o`, it finds no target entries for them. So, it checks for an appropriate implicit rule to apply. In this case, `make` selects the `.c.o` rule for building a `.o` file from a dependency file that has the same basename and a `.c` suffix.

make uses the order of appearance in the suffixes list to determine which dependency file and suffix rule to use. For instance, if there were both main.c and main.s files in the directory, make would use the .c.o rule, since .c is ahead of .s in the list.

First, make scans its suffixes list to see if the suffix for the target file appears. In the case of main.o, .o appears in the list. Next, make checks for an suffix rule to build it with, and a dependency file to build it from. The dependency file has the same basename as the target, but a different suffix. In this case, while checking the .c.o rule, make finds a dependency file named main.c, so it uses that rule.

The suffixes list is a special-function target named .SUFFIXES. The various suffixes are included in the definition for the SUFFIXES macro; the dependency list for .SUFFIXES is given as a reference to this macro:

Figure 5-6 *The Standard Suffixes List*

```
SUFFIXES=  .o .c .c~ .s .s~ .S .S~ .ln .f .f~ \
.F .F~ .l .l~ .mod .mod~ .sym .def .def~ .p .p~ \
.r .r~ .y .y~ .h .h~ .sh .sh~ .cps .cps~

.SUFFIXES:  $(SUFFIXES)
```

The following example shows a makefile for compiling a whole set of executable programs, each having just one source file. Each executable is to be built from a source file that has the same basename, and the .c suffix appended. For instance demo_1 is built from demo_1.c.

Like clean, all is a target name used by convention. It builds "all" the targets in its dependency list. Normally, all is the first target; make and make all are usually equivalent.

```
#   Makefile for a set of C programs, one source
#   per program.  The source file names have ".c"
#   appended.

CFLAGS= -O

.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

In this case, make does not find a suffix match for any of the targets (demo_1 through demo_5). So, it treats each as if it had a null suffix. It then searches for an suffix rule and dependency file with a valid suffix. In the case of demo_2, it would find a file named demo_2.c. Since there is a target entry for a .c rule, along with a corresponding .c file, make uses that rule to build demo_2 from demo_2.c.

To prevent ambiguity, when a target with a null suffix has an explicit dependency, make does not build it using a suffix rule. This makefile:

```
program: zap
zap:
```

produces no output:

```
hermes% make program
hermes%
```

When to Use Explicit Target Entries vs. Implicit Rules

Whenever you build a target from multiple dependency files, you must provide `make` with an explicit target entry that contains a rule for doing so. When building a target from a single dependency file, it is often convenient to use an implicit rule.

As the previous examples show, `make` readily compiles a single source file into a corresponding object file or executable. However, it has no built-in knowledge about how to link a list of object files into an executable program. Also, `make` only compiles those object files that it encounters in its dependency scan. It needs a starting point—a target for which each object file in the list (and ultimately, each source file) is a dependency.

So, for a target built from multiple dependency files, `make` needs an explicit rule that provides a collating order, along with a dependency list that accounts for its dependency files.

If each of those dependency files is built from just one source, you can rely on implicit rules for them.

Implicit Rules and Dynamic Macros

`make` maintains a set of macros dynamically, on a target-by-target basis. These macros are used quite extensively, especially in the definitions of implicit rules. So, it is important to understand what they mean.

They are:

Because they aren't explicitly defined in a makefile, the convention is to document dynamic macros with the $-sign prefix attached (in other words, by showing the macro reference).

`$@` The name of the current target.

`$?` The list of dependencies newer than the target.

`$<` The name of the dependency file, as if selected by `make` for use with an implicit rule.

`$*` The basename of the current target (the target name stripped of its suffix).

`$%` For libraries, the name of the member being processed. See *Building Object Libraries*, below, for more information.

Implicit rules make use of these dynamic macros in order to supply the name of a target or dependency file to a command line within the rule itself. For instance, in the `.c.o` rule, shown in the next example.

```
.c.o:
        $(COMPILE.c) $< $(OUTPUT_OPTION)
```

The macro `OUTPUT_OPTION` has an empty value by default. While similar to `CFLAGS` in function, it is provided as a separate macro intended for passing an argument to the `-o` compiler option to force compiler output to a given filename.

`$<` is replaced by the name of the dependency file (in this case the `.c` file) for the current target.

In the .c rule:

```
.c:
        $(LINK.c) $< -o $@
```

$@ is replaced with the name of the current target.

Because values for the $< and $* macros depend upon both the order of suffixes in the suffixes list, you may get surprising results when you use them in an explicit target entry. See *Suffix Replacement in Macro References* for a strictly deterministic method for deriving a filename from a related filename.

Dynamic Macro Modifiers

Dynamic macros can be modified by including F and D in the reference. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value. For example, with the target named /tmp/test, $(@D) has the value /tmp; $(@F) has the value test.

Dynamic Macros and the Dependency List: Delayed Macro References

Dynamic macros are assigned while processing any and all targets. They can be used within the target's rule as is, or in the dependency list by prepending an additional $ character to the reference. A reference beginning with $$ is called a *delayed* reference to a macro. For instance, the entry:

```
x.o y.o z.o: $$@.BAK
        cp $@.BAK $@
```

could be used to derive x.o from x.o.BAK, and so forth for y.o and z.o.

Dependency List Read Twice

This technique works because make reads the dependency list twice, once as part of its initial reading of the entire makefile, and again as it processes a target's dependencies. In each pass through the list, it performs macro expansion. Since the dynamic macros aren't defined in the initial reading, unless references to them are delayed until the second pass, they are expanded to null strings. The string $$ is a reference to the predefined macro '$'. This macro, conveniently enough, has the value '$'; when make resolves it in the initial reading, the string $$@ is resolved to $@. In dependency scan, when the resulting $@ macro reference has a value dynamically assigned to it, make resolves the reference to that value.

Note that make only evaluate the target-name portion of a target entry in the first pass. A delayed macro reference as a target name will produce incorrect results. The makefile:

```
NONE= none
all: $(NONE)

$$(NONE):
        @: this target's name isn't 'none'
```

produces the results shown below.

```
hermes% make
make: Fatal error: Don't know how to make target 'none'
```

Rules Evaluated Once

`make` evaluates the rule portion of a target entry only once per application of that command, at the time that the rule is executed. Here again, a delayed reference to a `make` macro will produce incorrect results.

No Transitive Closure for Suffix Rules

There is no transitive closure for suffix rules. If you had a suffix rule for building, say, a `.Y` file from a `.X` file, and another for building a `.Z` file from a `.Y` file, `make` would not combine their rules to build a `.Z` file from a `.X` file. You must specify the intermediate steps as targets, although their entries may have null rules:

```
trans.Z:
trans.Y:
```

In this example `trans.Z` will be built from `trans.Y` if it exists. Without the appearance of `trans.Y` as a target entry, `make` might fail with a "don't know how to build" error, since there would be no dependency file to use. The target entry for `trans.Y` guarantees that `make` will attempt to build it when it is out of date or missing. Since no rule is supplied in the makefile, `make` will use the appropriate implicit rule, which in this case would be the `.X.Y` rule. If `trans.X` exists (or can be retrieved from SCCS), `make` rebuilds both `trans.Y` and `trans.Z` as needed.

Adding Suffix Rules

Pattern-matching rules, which are described in the previous section, are often easier to use than suffix rules. The procedure for adding implicit rules is given here for compatibility with previous versions of `make`.

Although `make` supplies you with a number of useful suffix rules, you can also add new ones of your own. However, pattern-matching rules,[13] which are described in the next section, are to be preferred when adding new implicit rules. Unless you need to write implicit rules that are compatible with earlier versions of `make`, you may safely skip the remainder of this section, which describes the traditional method of adding implicit rules to makefiles.

Adding a suffix rule is a two-step process. First, you must add the suffixes of both target and dependency file to the suffixes list by providing them as dependencies to the `.SUFFIXES` special target. Because dependency lists

[13] Not available with earlier versions of `make`.

accumulate, you can add suffixes to the list simply by adding another entry for this target, for example:

```
.SUFFIXES: .ms .tr
```

Second, you must add a target entry for the suffix rule:

```
.ms.tr:
        troff -t -ms $< > $@
```

A makefile with these entries can be used to format document source files containing ms macros (.ms files) into troff output files (.tr files):

```
hermes% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Entries in the suffixes list are contained in the SUFFIXES macro. To insert suffixes at the head of the list, first clear its value by supplying an entry for the .SUFFIXES target that has no dependencies. This is an exception to the rule that dependency lists accumulate. You can clear a previous definition for this target by supplying a target entry with no dependencies and no rule like this:

```
.SUFFIXES:
```

You can then add another entry containing the new suffixes, followed by a reference to the SUFFIXES macro, as shown below.

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Pattern-Matching Rules: an Alternative to Suffix Rules

A *pattern-matching rule* is similar to an implicit rule in function. Pattern-matching rules are easier to write, and more powerful, because you can specify a relationship between a target and a dependency based on prefixes (including pathnames) and suffixes, or both. A pattern-matching rule is a target entry of the form:

tp%ts: *dp%ds*
        rule

where *tp* and *ts* are the optional prefix and suffix in the target name, respectively, *dp* and *ds* are the (optional) prefix and suffix in the dependency name, and % is a wild card that stands for a basename common to both.

`make` checks for pattern-matching rules ahead of suffix rules. While this allows you to override the standard implicit rules, doing so is not recommended.

If there is no rule for building a target, `make` searches for a pattern-matching rule, *before* checking for a suffix rule. If `make` can use a pattern-matching rule, it does so.

If the target entry for a pattern-matching rule contains no rule, `make` processes the target file as if it had an explicit target entry with no rule; `make` therefore searches for a suffix rule, attempts to retrieve a version of the target file from SCCS, and finally, treats the target as having a null rule (flagging that target as updated in the current run).

A pattern-matching rule for formatting a `troff` source file into a `troff` output file looks like:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

`make`'s Default Suffix Rules and Predefined Macros

The tables below show the standard set of suffix rules and predefined macros supplied to `make` in the default makefile, `/usr/include/make/default.mk`.

Table 5-1 `make`*'s Standard Suffix Rules*

Use	*Suffix Rule Name*	*Command Line(s)*
Assembly Files	.s.o	$(COMPILE.s) -o $@ $<
	.s.a	$(COMPILE.s) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
	.S.o	$(COMPILE.S) -o $@ $<
	.S.a	$(COMPILE.S) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
C Files	.c	$(LINK.c) -o $@ $< $(LDLIBS)
	.c.ln	$(LINT.c) $(OUTPUT_OPTION) -i $<
	.c.o	$(COMPILE.c) $(OUTPUT_OPTION) $<
	.c.a	$(COMPILE.c) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
FORTRAN 77 Files	.f	$(LINK.f) -o $@ $< $(LDLIBS)
	.f.o	$(COMPILE.f) $(OUTPUT_OPTION) $<
	.f.a	$(COMPILE.f) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
	.F	$(LINK.F) -o $@ $< $(LDLIBS)
	.F.o	$(COMPILE.F) $(OUTPUT_OPTION) $<
	.F.a	$(COMPILE.F) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%

Table 5-1 make's *Standard Suffix Rules— Continued*

Use	*Suffix Rule Name*	*Command Line(s)*
lex *Files*	.l	$(RM) $*.c $(LEX.l) $< > $*.c $(LINK.c) -o $@ $*.c $(LDLIBS) $(RM) $*.c
	.l.c	$(RM) $@ $(LEX.l) $< > $@
	.l.ln	$(RM) $*.c $(LEX.l) $< > $*.c $(LINT.c) -o $@ -i $*.c $(RM) $*.c
	.l.o	$(RM) $*.c $(LEX.l) $< > $*.c $(COMPILE.c) -o $@ $*.c $(RM) $*.c
Modula 2 *Files*	.mod .mod.o .def.sym	$(COMPILE.mod) -o $@ -e $@ $< $(COMPILE.mod) -o $@ $< $(COMPILE.def) -o $@ $<
NeWS	.cps.h	$(CPS) $(CPSFLAGS) $*.cps
Pascal	.p	$(LINK.p) -o $@ $< $(LDLIBS)
Files	.p.o	$(COMPILE.p) $(OUTPUT_OPTION) $<
Ratfor	.r	$(LINK.r) -o $@ $< $(LDLIBS)
Files	.r.o	$(COMPILE.r) $(OUTPUT_OPTION) $<
	.r.a	$(COMPILE.r) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $%
Shell *Scripts*	.sh	$(RM) $@ cat $< >$@ chmod +x $@
yacc *Files*	.y	$(YACC.y) $< $(LINK.c) -o $@ y.tab.c $(LDLIBS) $(RM) y.tab.c
	.y.c	$(YACC.y) $< mv y.tab.c $@
	.y.ln	$(YACC.y) $< $(LINT.c) -o $@ -i y.tab.c $(RM) y.tab.c
	.y.o	$(YACC.y) $< $(COMPILE.c) -o $@ y.tab.c $(RM) y.tab.c

Table 5-2 `make`'s *Predefined and Dynamic Macros*

Use	*Macro*	*Default Value*
Library Archives	AR	ar
	ARFLAGS	rv
Assembler Commands	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS) $(TARGET_ARCH)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) -target $(TARGET_ARCH:-%=%) -c
C Compiler Commands	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) -target $(TARGET_ARCH:-%=%)
C++ Compiler Commands	CCC	cc
	CCFLAGS	
	COMPILE.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS) -target $(TARGET_ARCH:-%=%)
FORTRAN 77 Compiler Commands	FC	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) $(TARGET_ARCH) -c
	LINK.f	$(FC) $(FFLAGS) $(TARGET_ARCH) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
Link Editor Command	LD	ld
	LDFLAGS	
`lex` *Command*	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
`lint` *Command*	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS) $(TARGET_ARCH)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS) $(TARGET_ARCH)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS) $(TARGET_ARCH)
NeWS	CPS	cps
	CPSFLAGS	
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) $(LDFLAGS)
`rm` *Command*	RM	rm -f
`yacc` *Command*	YACC	yacc
	YFLAGS	
	YACC.y	$(YACC) $(YFLAGS)
Suffixes List	SUFFIXES	.o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~ .cps .cps~
SCCS `get` *Command*	.SCCS_GET	sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ -G$@
	SCCSGETFLAGS	-s

5.3. Building Object Libraries

Libraries, Members and Symbols

An object library is a set of object files contained in an `ar` library archive.[14] Various languages make use of object libraries to store compiled functions of general utility, such as those in the C library.

`ar` reads in a set of one or more files to create a library. Each member contains the text of one file, preceded by a header. The member's header contains information from the file's directory entry, including the modification time. This allows `make` to treat the library member as a separate entity for dependency checking.

When you compile a program that uses functions from an object library (specifying the proper library either by filename, or with the `-l` option to `cc`), the link editor selects and links with the library member that contains a needed symbol.

You can use `ranlib` to generate a symbol table for a library of object files. `ld` requires this table in order to provide random access to symbols within the library—to locate and link object files in which functions are defined. You can also use `lorder` and `tsort` ahead of time to put members in calling order within the library. (See `lorder`(1) for details.) For very large libraries, it is a good idea to do both.

Library Members and Dependency Checking

`make` recognizes a target or dependency of the form

lib.a (*member* ...)

as a reference to a library member, or a space-separated list of members.[15] For example, the following target entry indicates that the library named `librpn.a` is built from members named `stacks.o` and `fifos.o`. The pattern-matching rule indicates that each member depends on a corresponding object file, and that object file is built from its corresponding source file using an implicit rule.

```
librpn.a: librpn.a(stacks.o fifos.o)
        ar rv $@ $?
        ranlib $@

librpn.a(%.o): %.o
        @true
```

When used with library-member notation, the dynamic macro `$?` contains the list of files that are newer than their corresponding members:

[14] See `ar`(1), `ar`(5), `lorder`(1), and `ranlib`(1) in the *Commands Reference Manual* for details about library archive files.

[15] Earlier versions `make` recognize this notation. However, only the first item in a parenthesized list of members was processed. In this version of make, all members in a parenthesized list are processed.

```
hermes% make
cc   -sun4 -c stacks.c
cc   -sun4 -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
ranlib librpn.a
```

Library Member Name-Length Limit

The name of an `ar` library member cannot exceed 15 characters. If a filename is longer than that, `ar` truncates the name of its corresponding member to the first 15 characters. If a library depends upon a member whose corresponding filename is too long, `make` attempts to match the name of the member to the first 15 characters of a file in the directory. *make* uses the first filename that matches as the file from which to build the member.

.PRECIOUS: Preserving Libraries Against Removal Due to Interrupts

Normally, if you interrupt `make` in the middle of a target, the target file is removed. For individual files this is a good thing, otherwise incomplete files with brand new modification times might be left in the directory. For libraries, which consist of several members, the story is different. It is often better to leave the library intact, even if one of the members is still out of date. This is especially true for large libraries, especially since a subsequent `make` run will pick up where the previous one left off—by processing the object file or member whose processing was interrupted.

`.PRECIOUS` is a special target that is used to indicate which files should be preserved against removal on interrupts; `make` does not remove targets that are listed as its dependencies. If you add the line:

```
.PRECIOUS: librpn.a
```

to the makefile shown above, run `make`, and interrupt the processing of `librpn.a`, the library is preserved.

Libraries and the $% Dynamic Macro

The `$%` dynamic macro is provided specifically for use with libraries. When a library member is the target, the member name is assigned to the `$%` macro. For instance, given the target `libx.a(demo.o)` the value of `$%` would be `demo.o`.

5.4. Maintaining Programs and Libraries With `make`

In previous sections you have learned how `make` can help compile simple programs and build simple libraries. This section describes some of `make`'s more advanced features for maintaining complex programs and libraries.

More about Macros

Macro definitions can appear on any line in a makefile; they can be used to abbreviate long target lists or expressions, or as shorthand to replace long strings that would otherwise have to be repeated. You can even use macros to derive lists of object files from a list of source files. Macro names are allocated as the makefile is read in; the value a particular macro reference takes depends upon the

most recent value assigned.[16] With the exception of conditional and dynamic macros, make assigns values in the order the definitions appear.

Embedded Macro References

Macro references can be embedded within other references,[17]

```
$(CPPFLAGS$(TARGET_ARCH))
```

in which case they are expanded from innermost to outermost. With the following definitions, make will supply the correct symbol definition for a Sun-3, or a Sun-4 system.

```
CPPFLAGS-sun3 = -DSUN3
CPPFLAGS-sun4 = -DSUN4
CPPFLAGS += $(CPPFLAGS$(TARGET_ARCH))
```

The += assignment appends the indicated string to any previous value for the macro.

Suffix Replacement in Macro References

make provides a mechanism for replacing suffixes of words that occur in the value of the referred-to macro.[18] A reference of the form:

$(*macro*:*old-suffix*=*new-suffix*)

is a *suffix replacement* macro reference. You can use a such a reference to express the list of object files in terms of the list of sources:

```
OBJECTS= $(SOURCES:.c=.o)
```

In this case, make replaces all occurrences of the .c suffix in words within the value with the .o suffix. The substitution is not applied to words for that do not end in the suffix given. The following makefile:

```
SOURCES= main.c data.c moon
OBJECTS= $(SOURCES:.c=.o)

all:
        @echo $(OBJECTS)
```

illustrates this very simply:

```
hermes% make
main.o data.o moon
```

[16] Actually, macro evaluation is a bit more complicated than this. Refer to *Passing Parameters to Nested* make *Commands* for more information.

[17] Not supported in previous versions of make.

[18] Although conventional suffixes start with dots, a suffix may consist of any string of characters.

Using `lint` with `make`

We encourage you to `lint` your C programs for easier debugging and maintenance. `lint` also checks for C constructs that are not considered portable across machine architectures. It can be a real help in writing portable C programs.

`lint`, the C program verifier,[19] is an important tool for forestalling the kinds of bugs that are most difficult and tedious to track down. These include uninitialized pointers, parameter-count mismatches in function calls, and nonportable uses of C constructs. As with the `clean` target, `lint` is a target name used by convention; it is usually a good practice to include it in makefiles that build C programs. `lint` produces output files that have been preprocessed through `cpp` and its own first (parsing) pass. These files characteristically end in the `.ln` suffix,[20] and can also be derived from the list of sources through suffix replacement:

```
LINTFILES= $(SOURCES:.c=.ln)
```

A target entry for the `lint` target might appear as:

```
lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

$(LINTFILES):
        $(LINT.c) $@ -i
```

There is an implicit rule for building each `.ln` file from its corresponding `.c` file, so there is no need for target entries for the `.ln` files. As sources change, the `.ln` files are updated whenever you run

```
make lint
```

Since the `LINT.c` predefined macro includes a reference to the `LINTFLAGS` macro, it is a good idea to specify the `lint` options to use by default (none in this case). Since `lint` entails the use of `cpp`, it is a good idea to use `CPPFLAGS`, rather than `CFLAGS` for compilation preprocessing options (such as `-I`). The `LINT.c` macro does not include a reference to `CFLAGS`.

Also, when you run `make clean` you will want to get rid of any `.ln` files produced by this target. It is a simple enough matter to add another such macro reference to a `clean` target.

Linking With System-Supplied Libraries

The next example shows a makefile that compiles a program that uses the `curses` and `termlib` library packages for screen-oriented cursor motion.

[19] See `lint` — a Program Verifier for C for more information.

[20] This is true for the Sun implementation, it may not be true for other versions of `lint`.

Figure 5-7 *Makefile for a C Program With System-Supplied Libraries*

```
#   Makefile for a C program with curses and termlib.
CFLAGS= -O
.KEEP_STATE:
functions: main.o data.o
        $(LINK.c) -o $@ main.o data.o -lcurses -ltermlib
lint: main.ln data.ln
        $(LINT.c) main.ln data.ln
main.ln data.ln:
        $(LINT.c) $@ -i
clean:
        rm -f functions main.o data.o main.ln data.ln
```

Since the link editor resolves undefined symbols as they are encountered, it is normally a good idea to place library references at the end of the list of files to link.

This makefile produces:

```
hermes% make
cc -O  -sun4 -c main.c
cc -O  -sun4 -c data.c
cc -O  -sun4 -o functions main.o data.o -lcurses -ltermlib
```

Compiling Programs for Debugging and Profiling

Compiling programs for debugging or profiling introduces a new twist to the procedure, and to the makefile. These variants are produced from the same source code, but are built with different options to the C compiler. The `cc` option to produce object code that is suitable for debugging is `-g`, and it is important to omit the `-O` option in this case. The `cc` options that produce code for profiling are `-O` and `-pg`.

Since the compilation procedure is the same otherwise, you *could* give make a definition for `CFLAGS` on the command line. Since this definition overrides the definition in the makefile, and `.KEEP_STATE` assures any command lines affected by the change are performed, the command:

```
make "CFLAGS= -O -pg"
```

produces the following results.

```
hermes% make "CFLAGS= -O -pg"
cc  -O -pg  -sun4 -c main.c
cc  -O -pg  -sun4 -c data.c
cc  -O -pg  -sun4 -o functions main.o data.o -lcurses -ltermlib
```

Of course, you may not want to memorize these options or type a complicated command like this, especially when you can put this information in the makefile. What is needed is a way to tell make how to produce a debugging or profiling variant, and some instructions in the makefile that tell it how. One way to do this might be to add two new target entries, one named `debug`, and the other named `profile`, with the proper compiler options hard-coded into the command line.

A better way would be to add these targets, but rather than hard-coding their rules, include instructions to alter the definition of `CFLAGS` depending upon which target it starts with. Then, by making each one depend on the existing target for `functions` `make` could simply make use of its rule, along with the specified options.

Instead of saying

```
make "CFLAGS= -g"
```

to compile a variant for debugging, you could say

```
make debug
```

The question is, how do you tell `make` that you want a macro defined one way for one target (and its dependencies), and another way for a different target?

Conditional Macro Definitions

A conditional macro definition[21] is a line of the form:

target-list `:=` *macro* `=` *value*

Each word in *target-list* may contain one `%` pattern; `make` must know which targets the definition applies to, so you can't use a conditional macro definition to alter a target name.

which assigns the given *value* to the indicated *macro* while `make` is processing the target named *target-name* and its dependencies. The following lines give `CFLAGS` an appropriate value for processing each program variant.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

Note that when you use a reference to a condition macro in the dependency list that reference must be delayed (by prepending a second `$`). Otherwise, `make` will expand the reference before the correct value has been assigned. When it encounters a (possibly) incorrect reference of this sort, `make` issues a warning.

Compiling Debugging and Profiling Variants

The following makefile produces optimized, debugging, or profiling variants of a C program, depending on which target you specify (the default is the optimized variant). Command dependency checking guarantees that the program and its object files will be recompiled whenever you switch between variants.

[21] Not available with previous versions of `make`.

Figure 5-8 *Makefile for a C Program with Alternate Debugging and Profiling Variants*

```
#   Makefile for a C program with alternate
#   debugging and profiling variants.

CFLAGS= -O

.KEEP_STATE:

all debug profile: functions

debug := CFLAGS = -g
profile := CFLAGS = -pg -O

functions: main.o data.o
        $(LINK.c) -o $@ main.o data.o -lcurses -ltermlib

lint: main.ln data.ln
        $(LINT.c) main.ln data.ln

clean:
        rm -f functions main.o data.o main.ln data.ln
```

The first target entry specifies three targets, starting with `all`.

Debugging and profiling variants aren't normally considered part of a finished program.

`all` traditionally appears as the first target in makefiles with alternate starting targets (or those that process a list of targets). It's dependencies are "all" targets that go into the final build, whatever that may be. In this case, the final variant is optimized. The target entry also indicates that `debug` and `profile` depend on `functions` (the value of `$(PROGRAM)`).

The next two lines contain conditional macro definitions for `CFLAGS`.

Next comes the target entry for `functions`. When `functions` is a dependency for `debug`, it is compiled with the `-g` option.

The next example applies a similar technique to maintaining a C object library.

Figure 5-9 *Makefile for a C Library with Alternate Variants*

```
#   Makefile for a C library with alternate
#   variants.

CFLAGS= -O

all debug profile: libpkg.a

.KEEP_STATE:
.PRECIOUS: libpkg.a

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

libpkg.a: libpkg.a(calc.o map.o draw.o)
        ar rv $@ $?
        ranlib $@

libpkg.a(%.o): %.o
        @true

lint: calc.ln map.ln draw.ln
        $(LINT.c) calc.ln map.ln draw.ln

clean:
        rm  -f libpkg.a calc.o map.o draw.o calc.ln \
        map.ln draw.ln
```

Maintaining Separate Program and Library Variants

The previous two examples are adequate when development, debugging and profiling are done in distinct phases. However they suffer from the drawback that all object files are recompiled whenever you switch between variants, which can result in unnecessary delays. The next two examples illustrate how all three variants can be maintained as separate entities.

To avoid the confusion that might result from having three variants of each object file in the same directory, you can place the debugging and profiling object files and executables in subdirectories. However, this requires a technique for adding the name of the subdirectory as a prefix to each entry in the list of object files.

Pattern-Replacement Macro References

A pattern-replacement macro reference is similar in form and function to a suffix replacement reference.[22] You can use a pattern-replacement reference to add or alter a prefix, suffix, or both, to matching words in the value of a macro. A pattern-replacement reference takes the form:

$(*macro*:*p*%*s*=*np*%*ns*)

[22] As with pattern-matching rules, pattern-replacement macro references aren't available in earlier versions of `make`.

where *p* is the existing prefix to replace (if any), *s* is the existing suffix to replace (if any), *np* and *ns* are the new prefix and new suffix, respectively, and % is a wild card. The pattern replacement is applied to all words in the value that match '*p*%*s*'. For instance:

```
SOURCES= old_main.c old_data.c moon
OBJECTS= $(SOURCES:old_%.c=new_%.o)

all:
        @echo $(NEW)
```

produces:

```
hermes% make
new_main.o new_data.o moon
```

You may use any number of % wild cards in the right-hand (replacement) side of the equal-sign, as needed. The following replacement:

```
...
NEW_OBJS= $(SOURCES:old_%.c=%/%.new)
```

would produce:

```
...
main/main.o data/data.o moon
```

Please note, however, that pattern-replacement macro references should not appear in the dependency line of the target entry for a pattern-matching rule. This produces a conflict, since make cannot tell whether the wild card applies to the macro, or to the target (or dependency) itself. With the makefile:

```
OBJECT= .o

x:
x.Z:
        @echo correct

%: %.$(OBJECT:%o=%Z)
```

it looks as if make should attempt to build x from x.Z. However, the pattern-matching rule is not recognized; make cannot determine which of the % characters in the dependency line to use in the pattern-matching rule.

Makefile for a Program with Separate Variants

`make` performs the rule in the `.INIT` target just after the makefile is read.

The following example shows a makefile for a C program with separately-maintained variants. First, the `.INIT` special target, creates the `debug` and `profile` subdirectories (if they don't already exist), which will contain the debugging and profiling object files and executables.

The variant executables are made to depend on the object files listed in the `VARIANTS.o` macro. This macro is given the value of `OBJECTS` by default; later on it is reassigned using a conditional macro definition, at which time either the `debug/` or `profile/` prefix is added. Executables in the subdirectories depend on the object files that are built in those same subdirectories.

Next, pattern-matching rules are added to indicate that the object files in both subdirectories depend upon source (`.c`) files in the working directory. This is the key step needed to allow all three variants to be built and maintained from a single set of source files.

Finally, the `clean` target has been updated to recursively remove the `debug` and `profile` subdirectories and their contents, which should be regarded as temporary. This is in keeping with the custom that derived files are to be built in the same directory as their sources, since the subdirectories for the variants are considered temporary.

Figure 5-10 *Makefile for Separate Debugging and Profiling Program Variants*

```
#   Simple makefile for maintaining separate debugging and
#   profiling program variants.

CFLAGS= -O

SOURCES= main.c rest.c
OBJECTS= $(SOURCES:%.c=$(VARIANT)%.o)
VARIANT=

functions profile debug: $$(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS)

debug := VARIANT = .debug/
debug := CFLAGS = -g
profile := VARIANT = .profile/
profile := CFLAGS = -O -pg

.KEEP_STATE:
.INIT: .profile .debug
.profile .debug:
        test -d $@ || mkdir $@

$$(VARIANT)%.o: %.c
        $(COMPILE.c) $< -o $@

clean:
        rm -r .profile .debug $(OBJECTS)
```

Makefile for a Library with Separate Variants

The modifications for separate library variants are quite similar:

Figure 5-11 *Makefile for Separate Debugging and Profiling Library Variants*

```
#   Makefile for maintaining separate library
#   variants.
CFLAGS= -O

SOURCES= main.c rest.c
LIBRARY= lib.a
LSOURCES= fnc.c

OBJECTS= $(SOURCES:%.c=$(VARIANT)%.o)
VLIBRARY= $(LIBRARY:%.a=$(VARIANT)%.a)
LOBJECTS= $(LSOURCES:%.c=$(VARIANT)%.o)
VARIANT=

program profile debug: $$(OBJECTS) $$(VLIBRARY)
        $(LINK.c) -o $@ $(OBJECTS) $(VLIBRARY)

lib.a .debug/lib.a .profile/lib.a: $$(LOBJECTS)
        ar rv $@ $?
        ranlib $@

$$(VLIBRARY)($$(VARIANT)%.o): $$(VARIANT)%.o
        @true

profile := VARIANT = .profile/
profile := CFLAGS = -O -pg

debug := VARIANT = .debug/
debug := CFLAGS = -g

KEEP_STATE:
.profile .debug:
        test -d $@ || mkdir $@

$$(VARIANT)%.o: %.c
        $(COMPILE.c) $< -o $@
```

While an interesting and useful compilation technique, this method for maintaining separate variants is a bit complicated. For clarity's sake it is omitted from subsequent examples.

Maintaining a Directory of Header Files

The makefile for maintaining an `include` directory of headers is really quite simple. Since headers consist of plain text, all that is needed is a target, `all`, that lists them as dependencies. Automatic SCCS retrieval takes care of the rest. If you use a macro for the list of headers, this same list can be used in other target entries.

```
#   Makefile for maintaining an include directory.
FILES.h= calc.h map.h draw.h
all: $(FILES.h)
clean:
        rm -f $(FILES.h)
```

Compiling and Linking With Your Own Libraries

When preparing your own library packages, it makes sense to treat each library as an entity that is separate from its header(s) and the programs that use it. Separating programs, libraries and headers into distinct directories often makes it easier to prepare makefiles for each type of module. And, it clarifies the structure of a software project.

It is not a good idea to have things pop up all over the file system as a result of running `make`.

A courteous and necessary convention of makefiles is that they only build files in the working directory, or in temporary subdirectories. Unless you are using `make` specifically to install files into a specific directory on an agreed-upon file system, it is regarded as very poor form for a makefile to produce output in another directory.

Building programs that rely on libraries in other directories adds several new wrinkles to the makefile. Up until now, everything needed has been in the directory, or else in one of the standard directories that are presumed to be stable. This is not true for user-supplied libraries that are part of a project under development.

Since these libraries aren't built automatically (there is no equivalent to hidden dependency checking for them), you must supply target entries for them. On the one hand, you need to assure the libraries you link with are up to date. On the other, you need to observe the convention that a makefile should only maintain files in the local directory. In addition, the makefile should not contain information duplicated in another.

Nested `make` Commands

The solution is to use a nested `make` command, running in the directory the library resides in, to rebuild it (according to the target entry in the makefile there).

The `MAKE` macro, which is set to the value "`make`" by default, overrides the `-n` option. Any command line in which it is referred to is executed, even though `-n` may be in effect. Since this macro is used to invoke `make`, and since the `make` it invokes inherits `-n` from the special `MAKEFLAGS` macro, `make` can trace a hierarchy of nested `make` commands with the `-n` option.

```
#   First cut entry for target in another
#   directory.
../lib/libpkg.a:
        cd ../lib ; $(MAKE) libpkg.a
```

The library is specified with a pathname relative to the current directory. In general, it is better to use relative pathnames. If the project is moved to a new root directory or machine, so long as its structure remains the same relative to that new root directory, all the target entries will still point to the proper files.

Within the nested make command line, the dynamic macro modifiers F and D come in handy, as does the MAKE predefined macro. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value.

The target entry can be rewritten as:

```
#   Second cut.
../lib/libpkg.a:
        cd $(@D); $(MAKE) $(@F)
```

Forcing A Nested make Command to Run

Because it has no dependencies, this target will only run when the file named ../lib/libpkg.a is missing. If the file is a library archive protected by .PRECIOUS, this could be a rare occurrence. The current make invocation neither knows nor cares about what that file depends on, nor should it. It is the nested invocation that decides whether and how to rebuild that file. After all, just because a file is present in the file system doesn't mean that it is up to date. This means that you have to force the nested make to run, regardless of the file's presence, by making it depend on another target with a null rule (and no extant file):

Figure 5-12 *Target Entry for a Nested* make *Command*

```
#   Reliable target entry for a nested make
#   command.
../lib/libpkg.a: FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:
```

In this way, make reliably cd's to the directory ../lib and builds libpkg.a if necessary, using instructions from the makefile found in that directory.

These lines are produced by the nested make run.

```
hermes% make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

The following makefile uses a nested make command to process local libraries that a program depends on.

Figure 5-13 *Makefile for C Program With User-Supplied Libraries*

```
#   Makefile for a C program with user-supplied
#   libraries and nested make commands.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o ../lib/libpkg.a
        $(LINK.c) -o $@ main.o data.o ../lib/libpkg.a -lcurses -ltermlib

../lib/libpkg.a: FORCE
        cd $(@D); $(MAKE) $(@F)
FORCE:

lint: main.ln data.ln
        $(LINT.c) main.ln data.ln

clean:
        rm -f functions main.o data.o main.ln data.ln
```

When `../lib/libpkg.a` is up to date, this makefile produces:

```
hermes% make
cc -O  -sun4 -c main.c
cc -O  -sun4 -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
cc -O  -sun4 -o functions main.o data.o ../lib/libpkg.a -lcurses -l termlib
```

The MAKEFLAGS Macro

Do not define MAKEFLAGS in your makefiles.

Like the MAKE macro, MAKEFLAGS is also a special case. It contains flags (that is, single-character options) for the `make` command. Unlike other FLAGS macros, the MAKEFLAGS value is a concatenation of flags, without a leading '-'. For instance the string, `eiknp` would be a recognized value for MAKEFLAGS, while, '`-f x.mk`' or '`macro=value`' would not.

If the MAKEFLAGS environment variable is set, `make` runs with the combination of flags given on the command line and contained in that variable.

The value of MAKEFLAGS is always exported, whether set in the environment or not, and the options it contains are passed to any nested `make` commands (whether invoked by `$(MAKE)`, `make` or `/usr/bin/make`). This insures you that nested `make` commands are always passed the options that the parent `make` was invoked with.

Macro Definitions and Environment Variables: Passing Parameters to Nested `make` Commands

With the exception of MAKEFLAGS,[23] `make` imports variables from the environment and treats them as if they were defined macros. In turn, `make` propagates

[23] and SHELL. The SHELL environment variable is neither imported nor exported in this version of `make`. See `make`(1) in the *SunOS Reference Manual*, for more information about the SHELL macro.

those environment variables and their values to commands it invokes, including nested make commands. Macros can also be defined as command line arguments, as well as the makefile. This can lead to name-value conflicts when a macro is defined in more than one place, and so, make has a fairly complicated precedence rule for resolving them.

First of all, conditional macro definitions always take effect within the targets (and their dependencies) for which they are defined.

If make is invoked with a macro-definition argument, that definition takes precedence over definitions given either within the makefile, or imported from the environment. (This does not necessarily hold true for nested make commands, however.) Otherwise, if you define (or redefine) a macro within the makefile, the most recent definition applies. The latest definition normally overrides the environment. Lastly, if the macro is defined in the default file and nowhere else, that value is used.

With nested make commands, definitions made in the makefile normally override the environment, but only for the makefile in which each definition occurs; the value of the corresponding environment variable is propagated regardless. Command-line definitions override both environment and makefile definitions, but only in the make run for which they are supplied. Although values from the command line are propagated to nested make commands, they are overridden both by definitions in the nested makefiles, and by environment variables imported by the nested make commands.

The -e option behaves more consistently. The environment overrides macro definitions made in any makefile, and command-line definitions are always used ahead of definitions in the makefile and the environment. One drawback to -e is that it introduces a situation in which information that is *not contained in the makefile* can be critical to the success or failure of a build.

To avoid these complications, when you want to pass a specific value to an entire hierarchy of make commands, run make -e in a subshell with the environment set properly:

```
hermes% (unsetenv MAKEFLAGS LDFLAGS; setenv CFLAGS -g ; make -e )
```

If you want to test out the cases yourself, you can use the following makefiles to illustrate the various cases.

```
# top.mk
MACRO= "Correct but unexpected."
top:
        @echo "------------------------------ top"
        echo $(MACRO)
        @echo "------------------------------"
        $(MAKE) -f nested.mk
        @echo "------------------------------ clean"
clean:
        rm nested
```

```
# nested.mk
MACRO=nested
nested:
        @echo "------------------------------ nested"
        touch nested
        echo $(MACRO)
        $(MAKE) -f top.mk
        $(MAKE) -f top.mk clean
```

Table 5-3 *Summary of Macro Assignment Order*

Without `-e`	*With* `-e` *in effect*
top-level `make` *command:*	
conditional definitions	conditional definitions
`make` command line	`make` command line
latest makefile definition	environment value
environment value	latest makefile definition
predefined value, if any	predefined value, if any
nested `make` *commands:*	
conditional definitions	conditional definitions
`make` command line	`make` command line
latest makefile definition	parent `make` cmd. line
environment variable	environment value
predefined value, if any	latest makefile definition
parent `make` cmd. line	predefined value, if any

Compiling Other Source Files

Compiling and Linking a C Program with Assembly Language Routines

The makefile in the next example maintains a program with C source files linked with assembly language routines.[24] There are two varieties of assembly source files, those that do not contain cpp preprocessor directives, and those that do. By convention, assembly source files without preprocessor directives have the .s suffix. Assembly sources that require preprocessing have the .S suffix.

Assembly sources are assembled to form object files in a fashion similar to that used to compile C sources. The object files can then be linked into a C program. make has implicit rules for transforming .s and .S files into object files, so a target entry for a C program with assembly routines need only specify how to link the object files. You can use the familiar cc command to link object files produced by the assembler:

ASFLAGS passes options for as to the .s.o and .S.o implicit rules.

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver:  c_driver.o s_routines.o S_routines.o
        cc -o driver c_driver.o s_routines.o S_routines.o
```

Note that the .S files are processed using the cc command, which invokes the C preprocessor cpp, and invokes the assembler implicitly.

Compiling lex and yacc Sources

lex and yacc produce C source files as output. Source files for lex end in the suffix .l, while those for yacc end in .y. When used separately, the compilation process for each is similar to that used to produce programs from C sources alone. There are implicit rules for compiling the lex or yacc sources into .c files; from there the files are further processed with the implicit rules for compiling object files from C sources. When these source files contain no #include statements, there is no need to keep the .c file, which in this simple case serves as an intermediate file. In this case one could use .l.o rule, or the .y.o rule, respectively, to produce the object files, and remove the (derived) .c files. For example, the makefile:

```
CFLAGS= -O
.KEEP_STATE:

all: scanner parser
scanner: scanner.o
parser: parser.o
```

produces the result shown below.

[24] Refer to the *Assembly Reference Manual* for more information about assembly language source files.

```
hermes% make
rm -f scanner.c
lex  -t scanner.l > scanner.c
cc -O  -sun4 -c scanner.c
cc -O  -sun4 scanner.c -o scanner
yacc  parser.y
cc -O  -sun4 -c y.tab.c -o parser.o
rm -f y.tab.c
yacc  parser.y
cc -O  -sun4 y.tab.c -o parser
rm -f y.tab.c
```

Things get to be a bit more complicated when you use `lex` and `yacc` in combination. In order for the object files to work together properly, the C code from `lex` must include a header produced by `yacc`. So, it may be necessary to recompile the C source file produced by `lex` when the `yacc` source file changes. In this case, it is better to retain the intermediate (`.c`) files produced by `lex`, as well as the additional `.h` file that `yacc` provides, so as to avoid running `lex` whenever the `yacc` source changes.

The following makefile maintains a program built from a `lex` source, a `yacc` source, and a C source file.

`yacc` produces output files named `y.tab.c` and `y.tab.h`. If you want the output files to have the same basename as the source file, you must rename them.

```
CFLAGS= -O
.KEEP_STATE:

a2z: c_functions.o scanner.o parser.o
        cc -o $@ c_functions.o scanner.o parser.o

scanner.c:

parser.c + parser.h: parser.y
        yacc -d parser.y
        mv y.tab.c parser.c
        mv y.tab.h parser.h
```

Since there is no transitive closure for implicit rules, you must supply a target entry for `scanner.c`. This entry bridges the gap between the `.l.c` implicit rule and the `.c.o` implicit rule, so that the dependency list for `scanner.o` extends to `scanner.l`. Since there is no rule in the target entry, `scanner.c` is built using the `.l.c` implicit rule.

The next target entry describes how to produce the `yacc` intermediate files. Because there is no implicit rule for producing both the header and the C source file using `yacc -d`, a target entry must be supplied that includes a rule for doing so.

Specifying Target Groups With the + Sign

In the target entry for `parser.c` and `parser.h`, the + sign separating the target names indicates that the entry is for a *target group*.[25] A target group is a set of files, all of which are produced when the rule is performed. Taken as a group, the set of files is what comprises the target. Without the + sign, each item listed would comprise a separate target. With a target group, `make` checks the modification dates separately against each target file, but performs the target's rule only once, if necessary, per `make` run.

Maintaining Shell Scripts with `make` and SCCS

Although a shell script is a plain text file, it must have execute permission to run. Since SCCS removes execute permission for files under its control, it is convenient to make a distinction between a shell script and it's "source" under SCCS. `make` has an implicit rule for deriving a script from its source. The suffix for a shell script source file is `.sh`. Even though the contents of the script and the `.sh` file are the same, the script has execute permissions, while the `.sh` file does not. `make`'s implicit rule for scripts "derives" the script from its source file, making a copy of the `.sh` file (retrieving it first, if necessary) and changing the mode of the resulting script file to allow execution. For example:

```
hermes% file script.sh
script.sh:      ascii text
hermes% make script
cat script.sh > script
chmod +x script
hermes% file script
script:         commands text
```

Running Tests with `make`

Shell scripts often come in handy for running tests, and performing other routine tasks that are either interactive, or don't require `make`'s dependency checking. Test suites, in particular, often entail providing a program with specific, repeatable input that a program might expect to receive from a terminal.

In the case of a library, a set of programs that exercise its various functions may be written in C, and then executed in a specific order, with specific inputs from a script. In the case of a utility program, there may be a set of benchmark programs that exercise and time its functions. In each of these cases, the commands to run each test can be incorporated into a shell script for repeatability and easy maintenance.

Once you have developed a test script that suits your needs, including a target to run it is easy. Although `make`'s dependency checking may not be needed within the script itself, you *can* use it to make sure that the program or library is updated before running those tests.

In the following target entry for running tests, `test` depends on the library named as a dependency to `all`. If the library is out of date, `make` rebuilds it and proceeds with the test. This insures that you always test with an up to date version:

[25] Not available with earlier versions of `make`.

```
test: all testscript
        set -x ; testscript > /tmp/test.\$\$

testscript: testscript.sh test_1 test_2 test_3

test_1 test_2 test_3: $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $< $(LIBRARY) $(SLIBS)
```

`test` also depends on `testscript`, which in turn depends on the three test programs. This assures that they too are up to date before `make` initiates the test procedure. `all` is built according to its target entry in the makefile; `testscript` is built using the `.sh` implicit rule; and the test programs are built using the rule in the last target entry, assuming that there is just one source file for each test program. (The `.c` implicit rule doesn't apply to these programs, because they must link with the proper libraries in addition to their respective `.c` files).

Escaped References to a Shell Variable

The string `\$\$` in the rule for `test` illustrates how to escape the dollar-sign from interpretation by `make`. `make` passes each `$`, to the shell, which expands the `$$` to its process ID. This technique allows each test to write to a unique temporary filename. The `set -x` command forces the shell to display the commands it runs on the terminal, which allows you to see the actual filename containing the results of the specific test.

Shell Command Substitutions

You can supply shell command substitutions within a rule as in the following example:

```
do:
        @echo `cat Listfile`
```

You can even place the backquoted expression in a macro:

```
DO= `cat Listfile`
do:
        @echo $(DO)
```

However, you can only use this form of command substitution within a rule.

Command Replacement Macro References

If you supply a shell command as the definition of a macro:

```
COMMAND= cat Listfile
```

you can use a *command replacement macro reference* to instruct `make` to replace the reference with the output of the command in the macro's value. This form of command substitution can occur anywhere within a makefile:

```
COMMAND= cat Listfile
$(COMMAND:sh): $$(@:=.c)
```

This example imports a list of targets from another file, and indicates that each target depends on a corresponding `.c` file.

As with shell command substitution, a command replacement reference evaluates to the standard output of the command. NEWLINE characters are converted to SPACE characters. The command is performed whenever the reference is encountered. The command's standard error is ignored. However, if the command returns a non-zero exit status, `make` halts with an error. A workaround for this is to append the `true` command to the command line:

```
COMMAND = cat Listfile ; true
```

Command Replacement Macro Assignment

A macro assignment of the form

cmd_macro`:sh` = *command*

assigns the standard output of the indicated *command* to *cmd_macro*; for instance:

```
COMMAND:sh = cat Listfile
$(COMMAND): $$(@:=.c)
```

is equivalent to the previous example. However, with the assignment form, the command is only performed once per `make` run. Again, only the standard output is used, NEWLINE characters are converted to SPACE characters, and a non-zero exit status halts `make` with an error.

Alternate forms of command replacement macro assignments are:

macro`:sh +=` *command*
: Append command output to the value of *macro*.

target `:=` *macro*`:sh =` *command*
: Conditionally define *macro* to be the output of *command* when processing *target* and its dependencies.

target `:=` *macro*`:sh +=` *command*
: Conditionally append the output of *command* to the value of *macro* when processing *target* and its dependencies.

5.5. Maintaining Software Projects

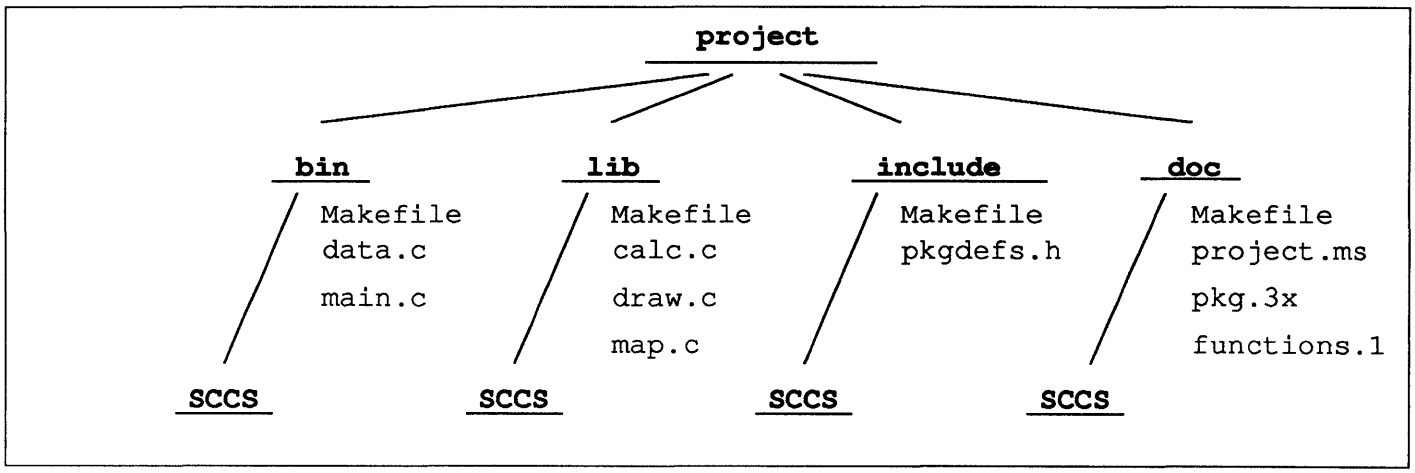
`make` is especially useful when a software project consists of a system of programs and libraries. By taking advantage of nested `make` commands, you can use it to maintain object files, executables, and libraries in a whole hierarchy of directories. You can use `make` in conjunction with SCCS, to assure that sources are maintained in a controlled manner, and that programs built from them are

consistent. This means that you can provide other programmers with duplicates of the directory hierarchy for simultaneous development and testing if you wish (although there are tradeoffs to consider).

You can use `make` to build the entire project and install final copies of various modules onto another filesystem for integration and distribution.

Organizing A Project for Ease of Maintenance

As mentioned earlier, one good way to organize a project is to segregate each major piece into its own directory. A project broken out this way usually resides within a single file-system or directory hierarchy. Header files could reside in one subdirectory, libraries in another, and programs in still another. Documentation, such as Reference Pages, may also be kept on hand in another subdirectory. Suppose that a project is composed of one executable program, one library that you supply, a set of headers for the library routines, and some documentation, as shown in the diagram below.



The makefiles in each subdirectory can be borrowed from examples in earlier sections, but something more is needed to manage the project as a whole. A carefully structured makefile in the root directory, the *root makefile* for the project, provides target entries for managing the project as a single entity.

As a project grows, the need for consistent, easy-to-use makefiles also grows. Macros and target names should have the same meanings no matter which makefile you are reading. Conditional macro definitions and compilation options for output variants should be consistent across the entire project.

Where feasible, a *template* approach to writing makefiles makes sense. This makes it easy for you keep track of how the project gets built. All you have to do to add a new type of module is to make a new directory for it, copy an appropriate makefile into that directory, and make a few edits. Of course, you also need to add the new module to the list of things to build in the root makefile.

Conventions for macro and target names, such as those used in the default makefile, should be instituted and observed throughout the project. Mnemonic names mean that although you may not remember the exact function of a target or value of a macro, you'll know the type of function or value it represents (and that's usually more valuable when deciphering a makefile anyway).

Using include Makefiles

One method of simplifying makefiles, while providing a consistent compilation environment, is to use make's

include *filename*

directive to read in the contents of a named makefile; if the named file is not present, make checks for a file by that name in /usr/include/make.

For instance, there is no need to duplicate the pattern-matching rule for processing troff sources in each makefile, when you can include it's target entry, as shown below.

```
SOURCES= doc.ms spec.ms
...
clean: $(SOURCES)
include ../pm.rules.mk
```

Here, make reads in the contents of the ../pm.rules.mk file, shown here:

```
#   pm.rules.mk
#
#   Simple "include" makefile for pattern-matching
#   rules.

%.tr: %.ms
        troff -t -ms $< > $@
%.nr: %.ms
        nroff -ms $< > $@
```

Installing Finished Programs and Libraries

When a program is ready to be released for outside testing or general use, you can use make to install it. Adding a new target and new macro definition to do so is easy:

```
DESTDIR= /proto/project/bin

install: functions
        -mkdir $(DESTDIR)
        cp functions $(DESTDIR)
```

A similar target entry can be used for installing a library, or a set of headers.

Building the Entire Project

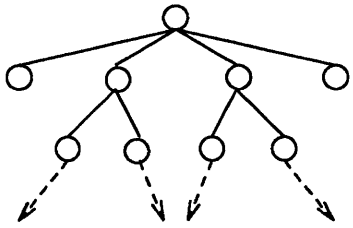
From time to time it is necessary to take a snapshot of the sources, and the object files that they produce. Building an entire project is simply a matter of invoking make successively in each subdirectory to build and install each module.

The following example shows how to use nested make commands to build a simple project.

```
#       Root makefile for a project.

TARGETS= all debug profile lint clean test install
SUBDIRS= bin include lib doc

$(TARGETS):
        $(MAKE) $(SUBDIRS) TARGET=$@

$(SUBDIRS): FORCE
        cd $@: $(MAKE) $(TARGET)

FORCE:
```

Maintaining Directory Hierarchies With Recursive Makefiles

If you extend your project hierarchy to include more layers:



chances are that not only will the makefile in each intermediate directory have to produce target files, but it will also have to invoke nested `make` commands for subdirectories of its own. Files in the current directory can sometimes depend on files in subdirectories, and their target entries need to depend on their counterparts in the subdirectories.

This means that the nested `make` command for each subdirectory should run before the command in the local directory does. One way to assure that the commands run in the proper order is to make a separate entry for the nested part, and another for the local part. If you add these new targets to the dependency list for the original target, its action will encompass them both.

Recursive Targets

Targets that encompass equivalent actions in both the local directory and in subdirectories are referred to as *recursive* targets.[26] A makefile with recursive targets is referred to as a *recursive* makefile.

In the case of `all`, the nested dependency can be named `all.nested`; the local dependency, `all.local`.

[26] Strictly speaking, any target that calls `make`, with its name as an argument, is recursive. However, here the term is reserved for the narrower case of targets that have both nested and local actions. Targets that only have nested actions are referred to as "nested" targets.

```
TARGETS= all debug profile lint clean test install
SUBDIRS= bin include lib doc
PROGRAM= functions

all: all.nested all.local

all.nested:
        $(MAKE) $(SUBDIRS) TARGET=$(TARGET)

$(SUBDIRS): FORCE
        cd $@; $(MAKE) $(TARGET)

all.local: $(PROGRAM)

FORCE:
```

Note that the "nested" target invokes `make` with the `all` target as an argument, *not* `all.nested`. The nested `make` must also be recursive, unless it is at the bottom of the hierarchy. In the makefile for a leaf directory (one with no subdirectories to descend into), you can simply comment out the rule for the nested target. This will halt any further descent.

Recursive `install` Targets

This same principle can be extended to all of the generic targets. The `install` target, however, is something of a special case. If the destination is a parallel directory hierarchy (such as when you are installing completed source code), the parent directories must be created before the destination subdirectories can be. This often means that the `make install` target in the current directory (which creates the destination directory if needed) must be performed before that in any subdirectory can succeed. So, `install.local` must appear ahead of `install.nested` in the dependency list for `install`:[27]

```
TARGETS= all debug profile lint clean test install
SUBDIRS= bin include lib doc
PROGRAM= functions

all: all.nested all.local
install: install.local install.nested

all.nested install.nested:
        $(MAKE) $(SUBDIRS) TARGET=$(@:%.nested=%)

$(SUBDIRS): FORCE
        cd $@; $(MAKE) $(TARGET)

all.local: $(PROGRAM)
install.local: $(PROGRAM)
```

[27] If the local target depends on files within a subdirectory, this may force `make` to descend into that subdirectory twice during a `make install` run.

Maintaining A Large Library as a Hierarchy of Subsidiaries

When maintaining a very large library, it is sometimes easier to break it up into smaller, subsidiary libraries, and use `make` to combine them into a complete package. Although you cannot combine libraries directly with `ar`, you can extract the member files from each subsidiary library, and then archive those files in another step, as shown below.

```
hermes% ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
hermes% ar xv liby.a
x - y1.o
x - y2.o
hermes% ar rv libz.a *.o
a - x1.o
a - x2.o
a - x3.o
a - y1.o
a - y2.o
ar: creating libz.a
```

A subsidiary library is maintained using a makefile in its own directory, along with the (object) files it is built from. The makefile for the complete library typically makes a symbolic link to each subsidiary archive, extracts their contents into a temporary subdirectory, and archives the resulting files to form the complete package.

The next example updates the subsidiary libraries, creates a temporary directory in which to extracted the files, and extracts them. It uses the * (shell) wild card within that temporary directory to generate the collated list of files. While filename wildcards are generally frowned upon, this use of the wild card is acceptable because the directory is created afresh whenever the target is built. This guarantees that it will contain only files extracted during the *current* `make` run.

In general, use of shell filename wildcards is considered to be bad form in a makefile. If you *do* use it, you need to take steps to insure that it excludes spurious files by isolating affected files in a temporary subdirectory.

The example relies on a naming convention for directories. The name of the directory is taken from the basename of the library it contains. For instance, if `libx.a` is a subsidiary library, the directory that contains it is named `libx`. It makes use of suffix replacements in dynamic-macro references to derive the directory name for each specific subdirectory. (You can verify yourself that this is necessary.)

It uses a shell `for` loop to successively extract each library, and a shell command substitution to collate the object files into proper sequence for linking (using `lorder` and `tsort`) as it archives them into the package. Finally, it removes the temporary directory and its contents.

```
#   Makefile for collating a library from subsidiaries.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS: libz.a

all: $(LIBRARY)

libz.a: libx.a liby.a
    -rm -rf tmp
    -mkdir tmp
    set -x ; for i in libx.a liby.a ; \
        do ( cd tmp ; ar x ../$$i ) ; done
    ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
    -ranlib $@
    -rm -rf tmp libx.a liby.a

libx.a liby.a: FORCE
    -cd $(@:.a=) ; $(MAKE) $@
    -ln -s $(@:.a=)/$@ $@
FORCE:
```

For the sake of clarity, this example omits support for alternate variants, as well as the targets for `clean`, `install`, and `test` (`lint` does not apply since the source files are in the subdirectories).

The `rm -f __.SYMDEF` command embedded in the collating line prevents a symbol table in a subsidiary (produced by running `ranlib` on that library) from being archived in this library.

Since the nested `make` commands build the subsidiary libraries before the currently library is processed, it is a simple matter to extend this makefile to account for libraries built from both subsidiaries and object files in the current directory. You need only add the list of object files to the dependency list for the library, and a command to copy them into the temporary subdirectory for collation with object files extracted from subsidiary libraries.

```
#   Makefile for collating a library from subsidiaries and local objects.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS: libz.a

all: libz.a

libz.a: libx.a liby.a map.o calc.o draw.o
        -rm -rf tmp
        -mkdir tmp
        -cp map.o calc.o draw.o tmp
        set -x ; for i in libx.a liby.a ; \
        do ( cd tmp ; ar x ../$$i ) ; done
        ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
        -ranlib $@
        -rm -rf tmp lix.a liby.a

libx.a liby.a: FORCE
        -cd $(@:.a=) ; $(MAKE) $@
        -ln -s $(@:.a=)/$@ $@
FORCE:
```

5.6. Closing Remarks about `make`

`make` has evolved into a powerful and flexible tool for consistently processing files that stand in a hierarchical relationship to one another. The methods and examples shown in this manual are intended to provide you with an exposure to the kinds of problems that lend themselves to solution with `make`. There is a large body of folklore about `make`; strong and varied opinions about its "best" use abound. This manual does not make the claim that any one approach or example is necessarily the best available. Compromises between clarity and functionality were made in many of the examples.

Also, there is considerable opinion both pro and against makefiles that use macros extensively. Some experts prefer to tailor makefiles for specific situations. Others prefer that all makefiles look the same and work the same way.

As procedures become more complicated, so do the makefiles that implement them. The trick is to know which approach will yield a reasonable makefile that works in a given situation. The examples are intended to give you a flavor for common situations, and some fairly straightforward methods to simplify them using `make`.

If a template approach is used in a project from the outset, chances are that custom makefiles that evolve from the templates will be more familiar, and therefore easier to understand, to integrate, to maintain, and more importantly, to *re-use.* After all, the less time you spend tinkering with the makefiles, the more time you have to develop your program or project.

6

`lint` — a Program Verifier for C

`lint` examines C source programs, detecting a number of bugs and obscurities. `lint` enforces the type rules of C more strictly than the C compiler. `lint` may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error-prone, constructions which nevertheless are, strictly speaking, legal.

`lint` accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between `lint` and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. `lint` takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of `lint`, gives an overview of its implementation, and gives some hints on writing machine-independent C code.

6.1. Using `lint`

Suppose there are two C source files, *file1 . c* and *file2 . c*, which are ordinarily compiled and loaded together. The command:

```
tutorial% lint  file1.c  file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. `lint` enforces the typing rules of C more strictly than the C compiler (for both historical and practical reasons) enforces them. The command:

```
tutorial% lint -p file1.c file2.c
```

produces, in addition to the types of messages described above, additional messages relating to portability of the programs to other operating systems and machines. As standardization efforts progress, `-p` may become less useful. Since many operating system implementations on a variety of hardware platforms are moving toward conformance with the System V Interface Definition (SVID), the X/OPEN Portability Guide, and the IEEE Std 1003.1-1988 (POSIX), the old definitions used by `-p` are less relevant.

Replacing the -p by -h produces messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying -hp gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. There is a summary of lint options in section lint Options.

6.2. A Word About Philosophy

Many of the facts which lint needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether exit is ever called is equivalent to solving the famous 'halting problem,' which is known to be recursively undecidable.

Thus, most of the lint algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, lint assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

lint tries to give information with a high degree of relevance. Messages of the form '*xxx* might be a bug' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which lint produces.

6.3. Unused Variables and Functions

As programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These 'errors of commission' rarely make working programs fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit extern statements but are never referenced; thus the statement:

```
extern  float  sin();
```

will evoke no comment if sin() is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the -x option to the lint invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The -v option is available to suppress the printing of complaints about unused arguments. When -v is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when `lint` is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The `-u` option may be used to suppress the spurious messages which might otherwise appear.

6.4. Set/Used Information

`lint` attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. `lint` detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a 'use,' since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that `lint` can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (for example, might contain at least two `goto`'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

6.5. Flow of Control

`lint` attempts to detect unreachable portions of the programs which it processes. It complains about unlabeled statements immediately following `goto`, `break`, `continue`, or `return` statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases `while(1)` and `for(;;)` as infinite loops. `lint` also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

`lint` has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to `exit` may cause unreachable code which `lint` does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement that `lint` does not complain about is a `break` statement that cannot be reached — programs generated by `yacc`, and especially `lex`, may have literally hundreds of unreachable `break` statements. The `-O` option in the C compiler often eliminates the resulting object code inefficiency. Thus, these unreached statements are of little importance — there is typically nothing the user can do about them, and the resulting messages would clutter up the `lint` output. If these messages are desired, `lint` can be invoked with the `-b` option.

6.6. Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function 'values' which are never returned. `lint` addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both:

```
return( expr );
```

and:

```
return;
```

statements results in the message

```
function name contains return( expr ) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a )
                return ( 3 );
        g ();
}
```

Notice that, if `a` tests false, `f()` calls `g()` and then returns with no defined return value; this triggers a complaint from `lint`. If `g()` never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the 'noise' messages produced by `lint`.

On a global scale, `lint` detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (for example, not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in 'working' programs; the desired function value just happened to have been computed in the function return register!

6.7. Type Checking

`lint` enforces the type checking rules of C more strictly than the compiler does. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (? :), and relational operators have this property; the argument of a `return` statement, and expressions used in initialization also suffer similar conversions. In these operations, `char`, `short`,

int, long, unsigned, float, and double types may be freely intermixed. The types of pointers must agree exactly, except that arrays of x's can, of course, be intermixed with pointers to x's.

The type checking rules also require that, in structure references, the left operand of the -> be a pointer to structure, the left operand of the '.' be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types float and double may be freely matched, as may the types char, short, int, and unsigned. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

6.8. Type Casts

The type casting feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where p is a character pointer. lint will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for lint to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The -c option controls the printing of comments about casts. When -c is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

6.9. Nonportable Character Use

In some implementations, characters are signed quantities, with a range from −128 to 127. In other C implementations, characters take on only positive values. Thus, lint will mark certain comparisons and assignments as being illegal or nonportable. For example, the fragment:

```
char c;
...
if( (c = getchar()) < 0 ) ...
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare c an integer, since getchar() is actually returning integer values. In any case, lint will say 'nonportable character comparison'.

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two-bit field declared of type `int` cannot hold the value 3, the problem disappears if the bitfield is declared to have type `unsigned`.

6.10. Assignments of Longs to Ints

Bugs may arise from the assignment of a `long` to an `int`, which may lose accuracy. This may happen in programs which have been incompletely converted to use `typedefs`. When a `typedef` variable is changed from `int` to `long`, the program can stop working because some intermediate results may be assigned to `int`'s, losing accuracy. Since there are a number of legitimate reasons for assigning `longs` to `ints`, the detection of these assignments is enabled by the `-a` option.

6.11. Strange Constructions

`lint` flags several perfectly legal, but somewhat strange, constructions — it is hoped that the messages encourage better code quality, clearer style, and may even point out bugs. The `-h` option is used to enable these checks. For example, in the statement:

```
*p++ ;
```

the * does nothing; this provokes the message 'null effect' from `lint`. The program fragment:

```
unsigned x ; if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test:

```
if( x > 0 ) ...
```

is equivalent to:

```
if( x != 0 )
```

which may not be the intended action. `lint` will say 'degenerate unsigned comparison' in these cases. If one says:

```
if( 1 != 0 ) ...
```

`lint` reports 'constant in conditional context', since the comparison of 1 with 0 gives a constant result.

Another construction detected by `lint` involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and `lint` encourages this by an appropriate message.

Finally, when the -h option is in force lint complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered to be bad style, usually unnecessary, and frequently a bug.

6.12. Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double-precision values may begin on any integer boundary. On the Honeywell 6000, double-precision values must begin on even word boundaries; thus, not all such assignments make sense. lint tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message 'possible pointer alignment problem' results from this situation whenever either the -p or -h options are in effect.

6.13. Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++] ;
```

will draw the complaint:

```
warning: i evaluation order undefined
```

6.14. Implementation

lint consists of two programs and a driver. The first program is a version of the Portable C Compiler, which is the basis of many C compilers, including Sun's. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the compilers do, lint produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of `lint`.

6.15. Portability

Many C programs have been successfully ported to a wide variety of operating systems, partly as a result of the `lint` features that increase portability. While there is no guarantee that a given C program will run unmodified within a different system environment, passing it through `lint` identifies and eliminates many potential portability problems.

For instance, uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as:

```
int a ;
```

outside of any function. The loader resolves these declarations, and sets aside only a single word of storage for `a`. Under the IBM implementations, this is not feasible, so each such declaration sets aside a word of storage called `a`. When loading or library editing takes place, this creates fatal conflicts which prevent the proper operation of the program. `lint` detects such multiple definitions if it is invoked with the `-p` option.

A related difficulty comes from the amount of information retained about external names during the loading process. On the SunOS system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on one system, but encounter loader problems on others. `lint -p` maps all external symbols to one case and truncates them to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the SunOS system are eight bit ASCII, while they are eight bit EBCDIC on the IBM, and nine bit ASCII on GCOS. Moreover, character strings go from high to low bit positions ('left to right') on GCOS and IBM, and low to high ('right to left') on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. `lint` is of little help here, except to option multi-character character constants.

Of course, the word sizes are different! This is less troublesome than might be expected, however. The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code

in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing:

```
x &= 0177700 ;
```

to clear the low order six bits of x. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing:

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed `unsigned`. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, `lint` is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of system utilities has been the inability to mimic essential system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, `lint` has been very helpful in moving the operating system and associated utility programs to other machines.

6.16. Shutting `lint` Up

There are occasions when the programmer is smarter than `lint`. There may be valid reasons for 'illegal' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by `lint` often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with `lint`, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to make `lint` recognize a number of words when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, `lint` directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the `lint` directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to `lint`, this can be asserted by placing the directive

```
/*NOTREACHED*/
```

just before that spot in the program. The `-v` option can be turned on for one function by the directive:

```
/*ARGSUSED*/
```

Complaints about variable numbers of arguments in calls to a function can be turned off by the directive:

```
/*VARARGS*/
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the `VARARGS` keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/*VARARGS2*/
```

checks the first two arguments and leaves the others unchecked. Finally, the directive:

```
/*LINTLIBRARY*/
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

6.17. Library Declaration Files

`lint` accepts certain library directives, such as:

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/*LINTLIBRARY*/
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The `VARARGS` and `ARGSUSED` directives can be used to specify features of the library functions.

`lint` library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file, but not used in a source file, draw no complaints. `lint` does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, lint checks the routines it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the –p option is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The –n option can be used to suppress all library checking.

6.18. Considerations When Using lint

lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause lint to miss errors which it should have caught. By contrast, if lint incorrectly complains about something that is correct, the programmer reports that immediately!

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of typedef is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with lint is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; lint concentrates on issues of portability, style, and efficiency. lint can afford to be wrong, since incorrectness and overconservatism are merely annoying, not fatal. The compiler can be fast since it knows that lint will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of lint, the desirable properties of universality and portability.

6.19. lint Options

The lint command currently has the form

```
tutorial% lint [-abchnpsuvx ] filename . . . library-descriptors . . .
```

The options are

a Report assignments of long to lint or shorter

b Report unreachable break statements

c Complain about questionable casts

h Perform heuristic checks

- `n` Do not do library checking
- `p` Perform portability checks
- `s` Same as `h` (for historical reasons)
- `u` Don't report unused or undefined externals
- `v` Don't report unused arguments
- `x` Report unused external declarations

7

Performance Analysis

Tools discussed in this chapter cover facilities for timing programs and getting performance analysis data. Some tools work only with the C programming language, while others will work on modules written in any language. Performance analysis tools provide a variety of levels of analysis from very simple timing of a command down to a statement-by-statement analysis of a program. You can select which level of granularity you like depending on the amount of detail and optimization you wish to perform. Here are the performance analysis tools available from the simplest to the most detailed:

`time` — A simple command (built in to the C shell) to display the time that a program takes. The C shell's built in `time` command display statistics about how a command uses the system resources as well as just the raw time consumed.

`prof` — Generates a profile for the modules in a program, showing which modules are using the time.

`gprof` — Generates not only a profile as for `prof`, but also generates a *call graph* showing what modules call which, and which modules are called by other modules. The call graph can sometimes point out areas where removing calls can speed up a program.

`tcov` — Generates a detailed statement-by-statement analysis of a C program.

7.1. `time` — Display Time Used by a Program

Two distinct versions of the `time` command exist in the Sun system. Here we discuss the `time` command that is built in to the C shell. The other `time` command is a program (in `/bin/time`) that you get when you use the Bourne shell.

As a first example, we show the `time` command being used to display statistics on the run-time of the `index.assist` program we've used in other examples in this manual. In all the examples shown here we direct the output from `index.assist` into `/dev/null`. Here is the simplest example of using `time`:

```
tutorial% time index.assist < index.entries > /dev/null
13.5u 0.8s 0:15 92% 3+19k 19+1io 0pf+0w
```

Now to explain the items in the display from the `time` command above.

The 13.5u means that this program used 13.5 seconds of *user* time — time spent in the application program itself. The 0.8s means that the program spent 0.8 seconds in the *system* — this is time spent in the operating system kernel on behalf of the program. The third field is the elapsed or wallclock time for the application. The percentage figure is the percent of the user and system time as a fraction of the elapsed time. The rest of the display is of lesser interest just now and is explained in more detail below.

Effects of Optimizer on Timing

Just for the sake of interest, let's see what effect the C optimizer has on the run time of this program — we make the program with the `-O` option and see what happens:

```
tutorial% time index.assist < index.entries > /dev/null
13.1u 1.4s 0:38 37% 3+19k 19+0io 1pf+0w
```

What has happened here? The optimized version takes longer to run! This demonstration tells us that simple timing is not so simple after all — in a multi-tasking system there are many other factors that can effect the simple timing. Note that the user time for the program is actually slightly less — 0.4 seconds less. But, the system time and the elapsed time are very different. These timings are affected by the load on the system. If we look at the last field in the time display, note that in the unoptimized version there were zero page faults, while in the optimized version there was one page fault. This is an indication that there was other activity in the system at the time the program was run and this other activity will adversely affect the elapsed time. There are two rules you can apply to this situation:

- Run such timing tests on a quiet system late at night. Make sure that 'late at night' is not midnight when a whole bunch of `cron` daemons start up.
- Run timing tests several times and take averages.

Controlling the display from the `time` Command

The `time` command built into the C shell has the capability of altering the information displayed under control of an environment variable. This is not true of `/bin/time` — the command you'd have to use if you were using the Bourne shell. Here is how to set up the `time` variable to control the time display.

You can control how the C shell times programs by setting the `time` variable in your `.login` or `.cshrc` file.

The `time` variable can be supplied with one or two values, such as `set time=3` or `set time=(3 "%E %P%")`.

Setting the `time` variable via a `set` command of the form:

```
set time=nnn
```

means that the shell displays a resource-usage summary for any command running for more than *nnn* CPU seconds.

Control Key Letters for the time Command

The second form controls exactly what resources are displayed. The character string can be any string of text with embedded control key-letters in it. A control key-letter is a percent sign (`%`) followed by a single *upper-case* letter. To print a percent sign, use two percent signs in a row. Unrecognized key-letters are simply printed. The control key-letters are:

Table 7-1 *Control Key Letters for the* `time` *Command*

Letter	*Description*
`D`	Average amount of unshared data space used in Kilobytes.
`E`	Elapsed (wallclock) time for the command.
`F`	Page faults.
`I`	Number of block input operations.
`K`	Average amount of unshared stack space used in Kilobytes.
`M`	Maximum real memory used during execution of the process.
`O`	Number of block output operations.
`P`	Total CPU time — U (user) plus S (system) — as a percentage of E (elapsed) time.
`S`	Number of seconds of CPU time consumed by the kernel on behalf of the user's process.
`U`	Number of seconds of CPU time devoted to the user's process.
`W`	Number of swaps.
`X`	Average amount of shared memory used in Kilobytes.

Default Timing Summary

The default resource-usage summary is a line of the form:

uuu.u`u` *sss.s*`s` *ee:ee* *pp*`%` *xxx*`+`*ddd*`k` *iii*`+`*ooo*`io` *mmm*`pf+`*ww*`w`

Table 7-2 *Default Timing Summary Chart*

Field	*Description*
uuu.u	user time (U),
sss.s	system time (S),
ee:ee	elapsed time (E),
pp	percentage of CPU time versus elapsed time (P),
xxx	average shared memory in Kilobytes (X),
ddd	average unshared data space in Kilobytes (D),
iii and *ooo*	the number of block input and output operations respectively (I and O),
mmm	number of page faults (F)
ww	number of swaps (W).

C shell `time` Command versus `/bin/time`

One final note on the `time` commands. As mentioned previously, there are two versions of `time`: the one built in to the C shell as described above, and the original Bourne shell `time` command which can be found in `/bin/time`.

The C shell `time` command does not time a command which is a component of a pipeline. This is what happens:

```
tutorial% echo timing a pipeline | time cat
timing a pipeline
```

whereas the Bourne shell `time` command gives completely different results:

```
tutorial% echo timing a pipeline | /bin/time cat
timing a pipeline
        0.8 real         0.0 user         0.1 sys
```

7.2. `prof` — Generate Profile of a Program

After simple timing, a *profile* of a program displays a finer level of analysis to assist in optimizing performance. Getting a profile is the next step after simple timing — more detailed analysis is provided by the *call-graph* profile and the *code coverage* tools described later in this chapter.

Taking the `index.assist` program from before as an example, let's make the program compiled for profiling. To compile a program for profiling, you use the `-p` option to the C compiler:

```
tutorial% make CFLAGS=-p
                .
                .
        messages from the make command
                .
                .
```

Now we can run the index.assist program as before. When a program is profiled, the results appear in a file called `mon.out` at the end of the run. Every time you run the program a new `mon.out` file is created, overwriting the old version. You then use the `prof` command to interpret the results of the profile, as shown by the example below.

```
tutorial% index.assist < index.entries > /dev/null
tutorial% prof index.assist
 %time  cumsecs  #call  ms/call  name
  19.4     3.28  11962     0.27  _compare_strings
  15.6     5.92  32731     0.08  _strlen
  12.6     8.06   4579     0.47  __doprnt
  10.5     9.84                  mcount
   9.9    11.52   6849     0.25  _get_field
   5.3    12.42    762     1.18  _fgets
   4.7    13.22  19715     0.04  _strcmp
   4.0    13.89   5329     0.13  _malloc
   3.4    14.46  11152     0.05  _insert_index_entry
   3.1    14.99  11152     0.05  _compare_entry
   2.5    15.41   1289     0.33  lmodt
   0.9    15.57    761     0.21  _get_index_terms
   0.9    15.73   3805     0.04  _strcpy
   0.8    15.87   6849     0.02  _skip_space
   0.7    15.99     13     9.23  _read
   0.7    16.11   1289     0.09  ldivt
   0.6    16.21   1405     0.07  _print_index
                    .
                    .
        everything else is insignificant
                    .
                    .
```

This display points out that most of the program's running time is spent in the routine that compares character strings to establish the correct place for the index entries, and that after that, the majority of the time is spent in the `_strlen` library routine — to find the length of a character string. If we wish to make any appreciable improvements to the program we must concentrate our efforts on the `compare_strings` function.

Interpreting Profile Display

Let's interpet the results of the profiling run though. The results appear under these column headings:

```
%time  cumsecs  #call  ms/call  name
```

Here's what the columns mean:

`%time` Percentage of the total run time of the program, that was consumed by this routine.

`cumsecs` A running sum of the number of seconds accounted for by this function and those listed above it. This information isn't really worth much — the important data comes from the percentage of total time and from the time consumed per call.

`#call` The number of times this routine was called.

`ms/call` How many milliseconds this routine consumed each time it was called.

`name` The name of the routine.

Now what advice can we derive from the profile data? Notice that the `compare_strings` function consumes nearly 20% of the total time. To improve the run time of `index.assist` we must either improve the algorithm that `compare_strings` uses, or we must cut down the number of calls. Not obvious from the flat profile is the information that `compare_strings` is heavily recursive — we get that fact from using the call graph profile described below. In this particular case, improving the algorithm also implies reducing the number of calls.

7.3. `gprof` — Generate a Call Graph Profile

While the flat profile described in the last section can provide valuable data for performance improvements, sometimes the data obtained is not sufficient to point out exactly where the improvements can be made. A more detailed analysis can be obtained by using the *call graph* profile that displays a list of which modules are called by other modules, and which modules call other modules. Sometimes, removing calls altogether can result in performance improvements.

Compiling with the `-pg` Option

Using the same `index.assist` program an example, let's make the program compiled for call-graph profiling. To compile a program for call-graph profiling, you use the `-pg` option to the C compiler:

```
tutorial% make CFLAGS=-pg
                .
                .
        messages from the make command
                .
                .
```

Now we can run the index.assist program as before. When a program is call-graph profiled, the results appear in a file called `gmon.out` at the end of the run. You then use the `gprof` command to interpret the results of the profile:

```
tutorial% index.assist < index.entries > /dev/null
tutorial% gprof index.assist
                .
                .
        voluminous output from the gprof command
                .
                .
```

Output from `gprof`

The output from `gprof` is really voluminous — it's usually intended that you take the summaries away and read them later. The output from `gprof` consists of the two major items listed below.

- The 'flat' profile. This is similar to the summary that the `prof` command supplies. `gprof` gives you slightly more information. The output from `gprof` contains an explanation of what the various parts of the summary mean, so you don't need to go look the things up in a manual.
- The full call-graph profile. There are some fragments of the output from the profiling run just below with some examples cf how to interpret them.

The output from `gprof` contains an explanation of what the various parts of the summary mean, so you don't need to go look the things up in a manual.

Interpreting Call Graph

Here is a fragment of the output from the `gprof` summary. Most of the output has been deleted from before and after the fragment. One thing that `gprof` does tell you is the granularity of the sampling:

```
granularity: each sample hit covers 4 byte(s) for 0.14% of 14.74 seconds
```

Then comes part of the call-graph profile itself:

```
                                     called/total       parents
index  %time    self descendents  called+self    name           index
                                     called/total       children

-----------------------------------------------

                0.00       14.47       1/1           start [1]
[2]     98.2    0.00       14.47       1         _main [2]
                0.59        5.70     760/760         _insert_index_entry [3]
                0.02        3.16       1/1           _print_index [6]
                0.20        1.91     761/761         _get_index_terms [11]
                0.94        0.06     762/762         _fgets [13]
                0.06        0.62     761/761         _get_page_number [18]
                0.10        0.46     761/761         _get_page_type [22]
                0.09        0.23     761/761         _skip_start [24]
                0.04        0.23     761/761         _get_index_type [26]
                0.07        0.00     761/820         _insert_page_entry [34]

-----------------------------------------------

                                   10392             _insert_index_entry [3]
                0.59        5.70     760/760         _main [2]
[3]     42.6    0.59        5.70     760+10392   _insert_index_entry [3]
                0.53        5.13   11152/11152       _compare_entry [4]
                0.02        0.01      59/112         _free [38]
                0.00        0.00      59/820         _insert_page_entry [34]
                                   10392             _insert_index_entry [3]

-----------------------------------------------
```

Noting that there are 761 lines of data in the input file to the `index.assist` program, here are some of the things we can determine from the call graph:

- `fgets` is called 762 times — one more than the number of lines in the input file. The last call to `fgets` returns an end-of-file.
- The `insert_index_entry` function is called 760 times from `main` — one less times than the number of lines. Why is this? The first index entry is inserted 'manually' in the `main` function when there are no previous index entries to insert.
- Note that in addition to the 760 times that `insert_index_entry` is called from `main`, `insert_index_entry` also calls *itself* the grand total of 10392 times — `insert_index_entry` is heavily recursive. Index entries appear in the input file in unsorted order and are sorted on the fly by inserting them into a binary tree.
- Note also that `compare_entry` (which is called from `insert_index_entry`) is called 11152 times, which is equal to 760+10392 times, so there is one call of `compare_entry` for every time that `insert_index_entry` is called. This is as it should be. If there was a discrepancy in the number of calls, we might suspect some problem in the program's logic.
- Notice the number of calls to the `insert_page_entry` and `free()` functions — `insert_page_entry` is called 820 times in total: 761 times from `main` while the program is building index nodes, and then `insert_page_entry` is called 59 times from `insert_index_entry`. This indicates that there are 59 index entries that are duplicated, so their page number entries are linked into a chain with the index nodes. The duplicate index entries are then freed, hence the 59 calls to `free()`.

7.4. `tcov` — Statement-Level Analysis

After a certain level of performance enhancements have been made, the profile data obtained from a program starts to look 'flat' and the granularity of the data collection makes further improvements difficult. At this point, you can use a tool that performs statement-by-statement analysis on a program, showing which statements are executed and how many times. This facility is called *code coverage*.

Code coverage can also be valuable in identifying areas of 'dead' code — areas of code that never get executed. Code coverage can also point out areas of code that are not being tested.

Compiling with the `-a` Option

Using the same `index.assist` program an example, let's make the program compiled for code coverage. To compile a program for code coverage, you use the `-a` option to the C compiler, as shown by the example below.

```
tutorial% make CFLAGS=-a
                .
                .
        messages from the make command
                .
                .
```

For every *thing*.c file you compile with the -a option, the C compiler generates a *thing*.d file — these are used by the code coverage program later in the analysis.

Using tcov

Now we can run the index.assist program as before. After a program has been run, you can then run tcov to get the summaries of execution counts for each statement in the program:

```
tutorial% index.assist < index.entries > /dev/null
tutorial% tcov *.c
```

Now, for every *thing*.c file you specify, tcov uses the *thing*.d file and generates a *thing*.tcov file containing and annotated listing of your code. The listing shows the number of times each source statement was executed. At the end of each *thing*.tcov file there is a short summary.

Below is a small fragment of the C code from one of the modules of index.assist — the module in question is the insert_index_entry function that's called so recursively.

```
                struct index_entry *
        insert_index_entry(node, entry)
11152 ->        struct   index_entry       *node;
                struct   index_entry       *entry;
        {

                int      result;
                int      level;

                result = compare_entry(node, entry);

                if (result == 0) {          /*  exact match  */
                                            /*  Place the page entry for the duplicate */
                                            /*  into the list of pages for this node */
   59 ->                insert_page_entry(node, entry->page_entry);
                        free(entry);
                        return(node);
                }

11093 ->        if (result > 0)             /*  node greater than new entry -- */
                                            /*  move to lesser nodes */
 3956 ->                if (node->lesser != NULL)
 3626 ->                        insert_index_entry(node->lesser, entry);
                        else {
  330 ->                        node->lesser = entry;
                                return (node->lesser);
                        }
                else                        /*  node less than new entry -- */
                                            /*  move to greater nodes */
 7137 ->                if (node->greater != NULL)
 6766 ->                        insert_index_entry(node->greater, entry);
                        else {
  371 ->                        node->greater = entry;
                                return (node->greater);
                        }
        }
```

Notice that the `insert_index_entry` function is indeed called 11152 times as we determined in the output from `gprof`. The numbers to the side of the C code show how many times each statement was executed.

`tcov` Summary

Below is the summary that `tcov` placed at the end of `build.index.tcov`.

```
          Top 10 Blocks

          Line      Count

           240      21563
           241      21563
           245      21563
           251      21563
           250      21400
           244      21299
           255      20612
           257      16805
           123      12021
           124      11962


    77   Basic blocks in this file
    55   Basic blocks executed
 71.43   Percent of the file executed

        439144   Total basic block executions
       5703.17   Average executions per basic block
```

8

m4 — a Macro Processor

m4 is a macro processor whose primary use has been as a front end for Ratfor in those cases where parameterless macros are not powerful enough. It has also been used for languages as disparate as C and COBOL. m4 is particularly suited for higher-level languages like FORTRAN, PL/I and C since macros are specified in a functional notation.

m4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The `#define` statement in C and the analogous `define` in Ratfor are examples of the basic facility provided by any macro processor, that is, replacement of text by other text.

The basic operation of m4 is to act as a filter between its input and its output. As the input is read, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then it macro is replaced by the text that has been assigned to it (*defining text*), and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the text before it is rescanned.

m4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can 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.

8.1. Using the m4 Command

The basic m4 command line looks like this:

```
m4  [filename ...]
```

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 to the standard output, which may be captured for subsequent processing using redirection:

```
m4 [filename ...] > outputfile
```

8.2. Defining Macros

The primary built-in function of m4 is `define`, which is used to define new macros. The input

```
define( name , value )
```

defines the string *name* as *value*. All subsequent occurrences of *name* will be replaced by *value*, unless *name* is redefined, or its definition is removed. Note that *name* must be alphanumeric, and must begin with a letter; the underscore character, _ is taken as a letter. The *value* argument is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example might be:

```
define(N, 100)
...
if (i > N)
```

defines N to be 100, and uses this "symbolic constant" in a later `if` statement.

The left parenthesis must immediately follow the word `define`, to signal that `define` has arguments. If a macro or built-in name is not followed immediately by '(', it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no parenthesis following it.

m4 divides its input into tokens, so a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains several N's.

Macros can be defined in terms of other macros. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In m4, the latter is true. M is translated to 100 as it is scanned, so changing N does not change M.

This behavior arises because m4 expands macro names into their defining text immediately. Here, that means that when the string N is seen while the arguments of define are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two alternatives. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when you ask for M later, you'll always get the value of N at that time (because the M will be replaced by N which will be replaced in turn by its value).

8.3. Quoting and Comments

The more general solution is to delay the expansion of the arguments of define by *quoting* them. Any text enclosed within the single-quote marks ` and ' is not expanded immediately, but merely has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, rather than the value of the N macro.

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 you want the word define to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining N:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the *N* in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

While this statement is ignored by m4, since you can only define macros with names that start with an alphabetical character or underscore, it obviously doesn't have the effect you wanted. To redefine N, you must delay the evaluation by quoting it:

```
define(N, 100)
...
define(`N', 200)
```

If the ` and ' characters are not convenient for some reason, the quote and end-quote characters can be changed with the built-in changequote function. For instance:

```
changequote([, ])
```

the left and right brackets the new quote and end-quote characters. You can restore the original characters with just **changequote**. There are two additional built-ins related to define. undefine removes the definition of a macro or built-in:

```
undefine(`N')
```

removes the definition of *N*. (Why are the quotes absolutely necessary?) Built-ins can be removed with undefine, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in ifdef provides a way to determine if a macro is currently defined. In particular, m4 pre-defines the name unix.

ifdef actually permits three arguments; if the name is undefined, the value of ifdef is then the third argument, as in

```
ifdef(`unix', on SunOS, not on SunOS)
```

Don't forget the quotes around the argument.

Comments in m4 are introduced by the # (sharp) character. All text from the # to the end of the line is taken as a comment and otherwise ignored.

8.4. Macros with Arguments

So far we have discussed the simplest form of macro processing — 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:

```
bump(x)
```

evaluates to

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through *$1* to *$9*. The macro name itself is *$0*, although that is less commonly used. Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* 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
```

$4 through *$9* are null, since no corresponding arguments were provided.

Leading unquoted SPACE's, TAB's, or NEWLINE's 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, but commas can be nested inside parentheses. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally *(b,c)*. And of course a bare comma or parenthesis can be inserted by quoting it.

8.5. Arithmetic Built-ins

`m4` provides two built-in functions for doing arithmetic on integers (only). The simplest is `incr`, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as "one more than N", write

```
define(N, 100)
define(N1, `incr(N)')
```

which defines `N1` as one more than the current value of `N`.

The more general mechanism for arithmetic is a built-in called `eval`, which is capable of arbitrary arithmetic on integers. `eval` provides the operators (in decreasing order of precedence), as shown in the table below.

Table 8-1 *Operators for the* `eval` *Built-In in* `m4`

Operator	*Meaning*
`unary + and -`	add and subtract
`**` *or* `^`	exponentiation
`*  /  %`	multiply, divide, and modulus
`+  -`	binary add and subtract
`==  !=  <  <=  >  >=`	equal, not equal, less than, less than or equal, greater than, greater than or equal
`!`	logical not
`&` *or* `&&`	logical and)
`|` *or* `||`	(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 (like 1>0) is 1, and false is 0. The precision in `eval` is 32 bits.

As a simple example, suppose we want `M` to be `2**N+1`. Then

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say, just a number); it usually gives the result you want, and is a good habit to get into.

8.6. File Manipulation

You can include a new file in the input at any time by the built-in function `include`:

```
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` (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in `include` cannot be accessed. To get some control over this, the alternate form `sinclude` can be used; `sinclude` ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of m4 to temporary files during processing, and output the collected material upon command. m4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as n. Diverting to this file is stopped by another divert command; in particular, divert or divert(0) resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and undivert with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of undivert is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros.

The built-in divnum returns the number of the currently active diversion. This is zero during normal processing.

8.7. Running SunOS Commands

You can run any SunOS command using the syscmd built-in. For example,

```
syscmd(date)
```

runs the *date* command. Normally syscmd would be used to create a file for a subsequent include.

To facilitate making unique file names, the built-in maketemp is provided, with specifications identical to the system function *mktemp*: a string of XXXXX in the argument is replaced by the process ID (*pid*) of the current process.

8.8. Conditionals

There is a built-in called ifelse which enables you to perform arbitrary conditional testing. In its simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If these are identical, ifelse returns the string *c*; otherwise it returns *d*. Thus we might define a macro called *compare* which compares two strings and returns "yes" or "no" according to whether they are the same or different.

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes, which prevent too-early evaluation of ifelse.

If the fourth argument is missing, it is treated as empty.

`ifelse` can actually have any number of arguments, and thus 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.

8.9. String Manipulation

The built-in `len` returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and `len((a,b))` is 5.

The built-in `substr` can be used to produce substrings of strings. `substr(s, i, n)` returns the substring of `s` that starts at the `i`th position (origin zero), and is `n` characters long. If `n` is omitted, the rest of the string is returned, so

```
substr('now is the time', 1)
```

evaluates to

```
ow is the time
```

If either `i` or `n` is out of range, various sensible things happen.

`index(s1, s2)` returns the index (position) in `s1` where the string `s2` occurs, or −1 if it doesn't occur. As with `substr`, the origin for strings is 0.

The built-in `translit` performs character transliteration.

```
translit(s, f, t)
```

modifies `s` by replacing any character found in `f` by the corresponding character in `t`. That is,

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If `t` is shorter than `f`, characters which don't have an entry in `t` are deleted; as a limiting case, if `t` is not present at all, characters in `f` are deleted from `s`. So

```
translit(s, aeiou)
```

deletes vowels from `s`.

There is also a built-in called `dnl` which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up `m4` output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add `dnl` to each of these lines, the newlines will disappear.

Another way to achieve this[28] is:

```
divert(-1)
        define(...)
        ...
divert
```

8.10. Printing

The built-in `errprint` writes its arguments to the standard error file. Thus you can say

```
errprint('fatal error')
```

`dumpdef` is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

8.11. Summary of Built-In `m4` Macros

Table 8-2 *Summary of Built-In* `m4` *Macros*

Built-In	*Description*
`changequote(L, R)`	Change left quote to L, right quote to R
`define(`*name*`,` *replacement*`)`	define *name* as *replacement*
`divert(`*number*`)`	Divert output to stream *number*
`divnum`	Return number of currently active diversions
`dnl`	Delete up to and including newline

[28] Thanks to J. E. Weythman.

Table 8-2 *Summary of Built-In* m4 *Macros— Continued*

Built-In	*Description*
`dumpdef(`*name'*, `*name'*, . . .)`	Dump specified definitions
`errprint(`*s*, *s*, . . .`)`	Write arguments *s* to standard error
`eval(`*numeric expression*`)`	Evaluate *numeric expression*
`ifdef(`*name'*, *true string*, *false string*`)`	Return *true string* if *name* is defined, *false string* if *name* is not defined
`ifelse(`*a*, *b*, *c*, *d*`)`	If *a* and *b* are equal, return *c*, else return *d*
`include(`*file*`)`	Include contents of *file*
`incr(`*number*`)`	Increment *number* by 1
`index(`*s1*, *s2*`)`	Return position in *s1* where *s2* occurs, or –1 if no occurrence
`len(`*string*`)`	Return length of *string*
`maketemp(. . .XXXXX. . .)`	Make a temporary file
`sinclude(`*file*`)`	Include contents of *file* — ignored and continue if *file* not found.
`substr(`*string*, *position*, *number*`)`	Return substring of *string* starting at *position* and *number* characters long
`syscmd(`*command*`)`	Run *command* in the system
`translit(`*string*, *from*, *to*`)`	Transliterate characters in *string* from the set specified by *from* to the set specified by *to*
`undefine(`*name'*`)`	Remove *name* from the list of definitions
`undivert(`*number*,*number*, . . .`)`	Append diversion *number* to the current diversion

9

lex — a Lexical Analyzer Generator

lex is a program generator designed for lexical processing of character input streams. lex accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. The regular expressions are specified by the programmer in the source specifications given to lex. The lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the programmer are executed. The lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by lex, the corresponding fragment is executed.

The programmer supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general-purpose programming language employed for the programmer's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the programmer's freedom to write actions is unimpaired. This avoids forcing the programmer who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by lex. The program fragments written by the programmer are executed in the order in which the corresponding regular expressions occur in the input stream.

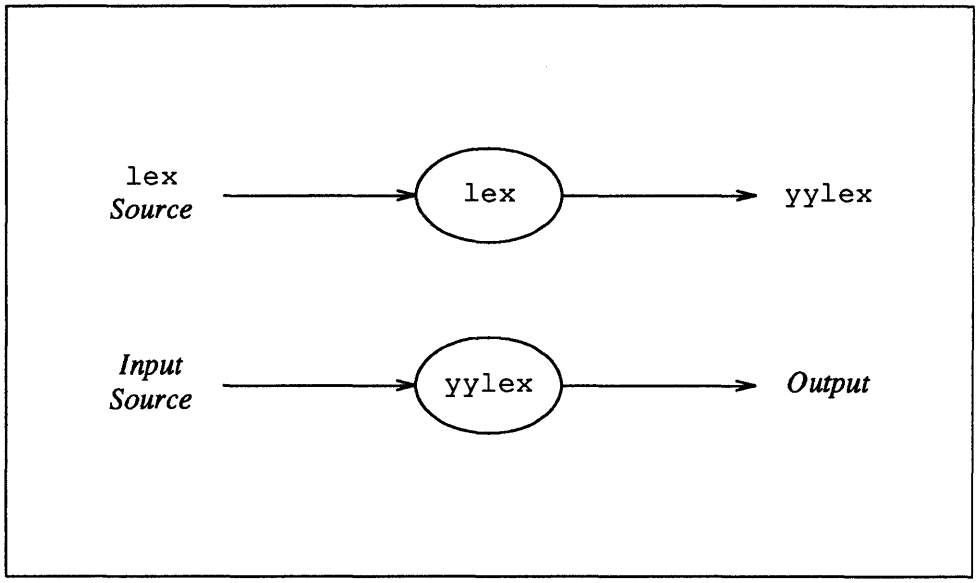
The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream is then backed up to the end of the current partition, so that the programmer has general freedom to manipulate it.

lex is designed to simplify interfacing with yacc, which is described in the next chapter.

`lex` is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called 'host languages.' Just as general-purpose languages can produce code to run on different computer hardware, `lex` can write code in different host languages. The host language is used for the output code generated by `lex` and also for the program fragments added by the programmer. Compatible run-time libraries for the different host languages are also provided. This makes `lex` adaptable to different environments and different programmer. Each application may be directed to the combination of hardware and host language appropriate to the task, the programmer's background, and the properties of local implementations.

`lex` turns the programmer's expressions and actions (called `source` in this document) into the host general-purpose language; the generated program is named `yylex`. The `yylex` program recognizes expressions in a stream (called `input` in this document) and performs the specified actions for each expression as it is detected — see Figure 9-1 below.

Figure 9-1 *An overview of* `lex`



For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates 'one or more ...'; and the $ indicates 'end-of-line'. No action is specified, so the program generated by `lex` (`yylex`) ignores these characters. Everything else is

copied to the output stream. To change any remaining string of blanks or tabs to a single blank, add another rule:

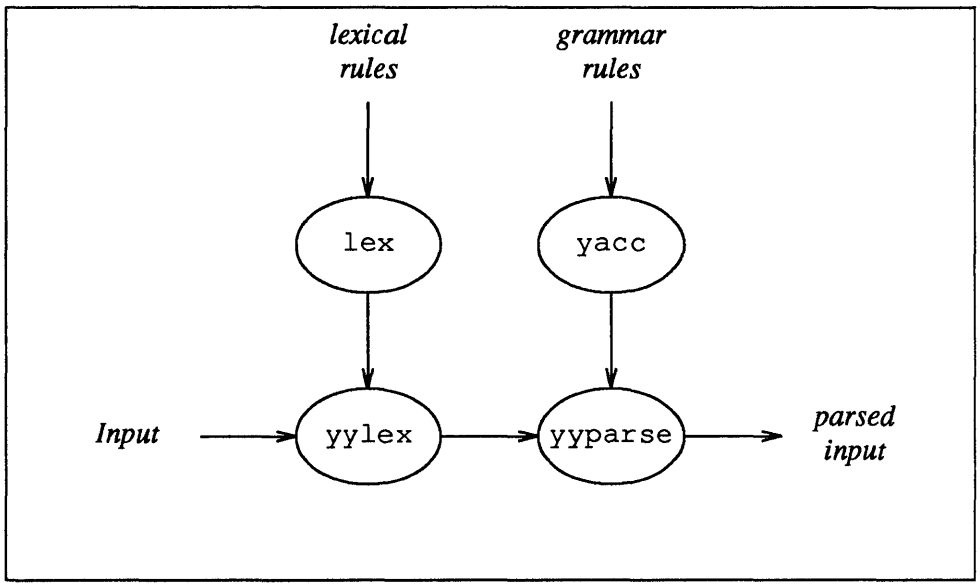
```
%%
[ \t]+$ ;
[ \t]+  printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the ends of lines, and the second rule all remaining strings of blanks or tabs.

lex can also be used with a parser generator to perform the lexical analysis phase.

lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface lex and yacc lex programs recognize only regular expressions; yacc writes parsers that accept a large class of context-free grammars, but require a lower-level analyzer to recognize input tokens. Thus, a combination of lex and yacc is often appropriate. When used as a preprocessor for a later parser generator, lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 9-2. Additional programs, written by other generators or by hand, can be added easily to programs written by lex.

Figure 9-2 lex *with* yacc



yacc programmers will realize that the name yylex is what yacc expects its lexical analyzer to be named, so that the use of this name by lex simplifies interfacing.

`lex` generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a `lex` program to recognize and partition an input stream is proportional to the length of the input. The number of `lex` rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by `lex`.

In the program written by `lex`, the programmer's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the programmer to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

`lex` is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for `ab` and another for `abcdefg`, and the input stream is `abcdefh`, `lex` recognizes `ab` and leave the input pointer just before `"cd..."` Such backup is more costly than processing simpler languages.

9.1. `lex` Source

The general format of `lex` source is:

```
{definitions }
%%
{rules}
%%
{programmer subroutines }
```

where the definitions and the programmer subroutines are often omitted. The second `%%` is optional, but the first is required to mark the beginning of the rules. The absolute minimum `lex` program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of `lex` programs shown above, the *rules* represent the programmer's control decisions; they are a table, in which the left column contains *regular expressions* (see section 9.2) and the right column contains *actions*, program fragments to be executed when the expressions

```
integer printf("found keyword INT");
```

to look for the string `integer` in the input stream and print the message 'found keyword INT' whenever it appears. In this example the host procedural language is C and the C library function `printf()` is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is

merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. `lex` rules such as

```
colour  printf("color");
mechanise       printf("mechanize");
petrol  printf("gas");
```

would be a start. These rules are not quite enough, since the word `petroleum` would become `gaseum`; a way of dealing with this is described later.

9.2. `lex` Regular Expressions

The definitions of regular expressions are very similar to those in the editors *ex*(1) and *vi*(1). A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string `integer` wherever it appears and the expression

```
a57D
```

looks for the string `a57D`.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape must be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string `xyz++` when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the programmer can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as

explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction `[abc]` matches a single character, which may be `a`, `b`, or `c`. Within square brackets, most operator meanings are ignored. Only three characters are special: \, –, and ^. The – character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation-dependent and generates a warning message. For example, `[0-z]` in ASCII is many more characters than it is in EBCDIC. If it is desired to include the character – in a character class, it should be first or last, thus:

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the system's character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; and

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

(period) is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either ac or abc.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab | cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab | cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab | cd+)?(ef)*
```

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only be matched at the beginning of a line This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression is only be matched at the end of a line (when immediately followed by newline).

The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string `ab`, but only if it is followed by `cd`. Thus

```
ab$
```

is the same as

```
ab/\n.
```

Left context is handled in `lex` by *start conditions* as explained in section 9.9 — *Left Context-Sensitivity*. If a rule is only to be executed when the `lex` automaton interpreter is in start condition `x`, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered 'being at the beginning of a line' to be start condition `ONE`, then the ^ operator would be equivalent to

```
<ONE>.
```

Start conditions are explained more fully below.

Repetitions and Definitions

The operators { } specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named `digit` and inserts it at that point in the expression. The definitions are given in the first part of the `lex` input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of `a`.

Finally, initial `%` is special, being the separator for `lex` source segments.

9.3. `lex` Actions

When an expression written as above is matched, `lex` executes the corresponding action. This section describes some features of `lex` which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the `lex` programmer who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When `lex` is being used with `yacc`, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action does this. A frequent rule is

```
[ \t\n] ;
```

which ignores the three spacing characters (blank, tab, and newline).

Another easy way to avoid writing actions is the action character |, which indicates that the action to be used for this rule is the action given for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result. The quotes around \n and \t are not required.

Actual Text that Matched

In more complex actions, the programmer often wants to know the actual text that matched some expression like `[a-z]+`. `lex` leaves this text in an external character array named `yytext`.
Thus, to print the name found, a rule like

```
[a-z]+ printf("%s", yytext);
```

prints the string in `yytext`. The C function `printf` accepts a format argument and data to be printed; in this case, the format is 'print string' (% indicating data conversion, and `s` indicating string type), and the data are the characters in `yytext`. So this just places the matched string on the output. This action is so common that it may be written as `ECHO`:

```
[a-z]+ ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches `read()` it normally matches the instances of `read` contained in `bread` or `readjust`; to avoid this, a rule of the form `[a-z]+` is needed. This is explained further below.

Length of Matched Text

Sometimes it is more convenient to know the end of what has been found; hence `lex` also provides a count `yyleng` of the number of characters matched. To count both the number of words and the number of characters in words in the input, the programmer might write

```
[a-zA-Z]+     {words++; chars += yyleng;}
```

which accumulates in `chars` the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1].
```

yymore *and* yyless

Occasionally, a `lex` action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, `yymore()` can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in `yytext`. Second, `yyless(n)` may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument `n` indicates the number of characters to be retained in `yytext`. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write:

```
\"[^"]* {
        if (yytext[yyleng-1] == '\\')
             yymore();
        else
             ... normal programmer processing
        }
```

which, when faced with a string such as `"abc\"def"` first matches the five characters `"abc\`; then the call to `yymore()` tacks the next part of the string, `"def`, onto the end. Note that the final quote terminating the string should be picked up in the code labeled 'normal processing'.

The function `yyless()` might be used to reprocess text in various circumstances. Consider the problem of resolving (in old-style C) the ambiguity of '=–a'. Suppose it is desired to treat this as '=– a' but print a message. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as '=–'. Alternatively it might be desired to treat this as '= –a'. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

performs the other interpretation. Note that the expressions for the two cases might more easily be written:

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of '=-3', however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, `lex` also permits access to the I/O routines it uses. They are:

1. `input()` which returns the next input character;
2. `output(c)` which writes the character `c` on the output; and
3. `unput(c)` pushes the character `c` back onto the input stream to be read later by `input()`.

By default these routines are provided as macro definitions, but the programmer can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to transmit input or output to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by `input` must mean end of file; and the relationship between `unput` and `input` must be retained or the `lex` lookahead will not work. `lex` does not look ahead at all if it does not have to, but every rule ending in `+ * ?` or `$` or containing `/` implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See section 9.10 for a discussion of the character set used by `lex`. The standard `lex` library imposes a 100-character limit on backup.

Another `lex` library routine that the programmer will sometimes want to redefine is `yywrap()` which is called whenever `lex` reaches an end-of-file. If `yywrap` returns a 1, `lex` continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the programmer should provide a `yywrap` which arranges for new input and returns 0. This instructs `lex` to continue processing. The default `yywrap` always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through `yywrap`.
In fact, unless a private version of `input()` is supplied a file containing nulls cannot be handled, since a value of 0 returned by `input` is taken to be end-of-file.

9.4. Ambiguous Source Rules

`lex` can handle ambiguous specifications. When more than one expression can match the current input, `lex` chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ... ;
[a-z]+  identifier action ... ;
```

to be given in that order. If the input is `integers`, it is taken as an identifier, because `[a-z]+` matches 8 characters, while `integer` matches only 7. If the input is `integer`, both rules match 7 characters, and the keyword rule is selected because it was given *first*. Anything shorter (for example, `int`) will not match the expression `integer`, and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression matches

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, stops after `'first'`. The consequences of errors like this are mitigated by the fact that the `.` operator does not match newline. Thus expressions like `.*` stop on the current line. Don't try to defeat this with expressions like `[.\n]+` or equivalents; the `lex` generated program will try to read the entire input file, causing internal buffer overflows.

Note that `lex` is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both `she` and `he` in an input text. Some `lex` rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides he and she. Remember that '.' does not include newline. Since she includes he, lex will normally not recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the programmer would like to override this choice. The action REJECT means 'go do the next alternative.' It executes whatever rule was second choice after the current rule. The position of the input pointer is adjusted accordingly. Suppose the programmer really wants to count the included instances of he:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression is then counted. In this example, of course, the programmer could note that she includes he but not vice versa, and omit the REJECT action on he; in other cases, however, it would not be possible *a priori* to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is ab, only the first rule matches, and on ad only the second matches. The input string accb matches the first rule for four characters and then the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and the first rule for three.

In general, REJECT is useful whenever the purpose of lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word the is considered to contain both th and he. Assuming a two-dimensional array named digram to be incremented, the appropriate source is shown below.

```
%%
[a-z][a-z]   {digram[yytext[0]][yytext[1]]++; REJECT;}
.        ;
\n       ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

9.5. lex Source Definitions

Remember the format of the lex source:

```
{definitions}
%%
{rules}
%%
{programmer routines}
```

So far only the rules have been described. The programmer needs additional options, though, to define variables for use in his program and for use by lex. These can go either in the definitions section or in the rules section.

Remember that lex is turning the rules into a program. Any source not intercepted by lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a lex rule or action which begins with a blank or tab is copied into the lex-generated program. Such source input prior to the first %% delimiter is external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by lex which contains the actions. This material must look like program fragments, and should precede the first lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only the delimiters %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the lex output.

Definitions intended for lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define lex substitution strings. The format of such lines is

```
name translation
```

and it associates the string given as a translation with the name. The name and

translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be invoked by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D       [0-9]
E       [DEde][-+]?{D}+
%%
{D}+    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}         printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as `35.EQ.I`, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within `lex` itself for larger source programs. These possibilities are discussed below under section 9.11 — *Summary of Source Format.*

9.6. Using `lex`

There are two steps in compiling a `lex` source program. First, the `lex` source must be turned into a generated program in the host general-purpose language. Then this program must be compiled and loaded, usually with a library of `lex` subroutines. The generated program is on a file named `lex.yy.c`. The I/O library is defined in terms of the C standard library in section 3 of the *SunOS Reference Manual.*

The `lex` library is accessed by the loader flag `-ll`.
So an appropriate set of commands is:

```
tutorial% lex source
tutorial% cc lex.yy.c -ll
```

The resulting program is placed on the usual file `a.out` for later execution. To use `lex` with `yacc` see below. Although the default `lex` I/O routines use the C standard library, the `lex` automata themselves do not do so; if private versions of `input`, `output`, and `unput` are given, the library can be avoided. `lex` has several options which are described in the `lex`(1) manual page.

9.7. lex and yacc

If you want to use `lex` with `yacc`, note that what `lex` writes is a program named `yylex()`, the name required by `yacc` for its analyzer. Normally, the default main program in the `lex` library calls this routine, but if `yacc` is loaded, and its main program is used, `yacc` calls `yylex()`.

In this case each `lex` rule should end with

```
return(token);
```

to return the appropriate token value.

An easy way to get access to `yacc`'s names for tokens is to compile the `lex` output file as part of the `yacc` output file by placing the line

```
# include "lex.yy.c"
```

in the last section of `yacc` input. Supposing the grammar to be named 'good' and the lexical rules to be named 'better' the command sequence can just be:

```
tutorial% yacc good
tutorial% lex better
tutorial% cc y.tab.c -ll
tutorial%
```

The `lex` and `yacc` programs can be generated in either order.

9.8. Examples

As a trivial problem, consider copying an input file while adding 3 to every non-negative number divisible by 7. Here is a suitable `lex` source program

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
             printf("%d", k+3);
        else
             printf("%d",k);
        }
```

to do just that. The rule `[0-9]+` recognizes strings of digits; `atoi()` converts the digits to binary and stores the result in `k`.

The operator `%` (remainder) is used to check whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as shown below.

```
%%
        int k;
-?[0-9]+{
        k = atoi(yytext);
        printf("%d", k%7 == 0 ? k+3 : k);
        }
-?[0-9.]+       ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a '.' or preceded by a letter are picked up by one of the last two rules, and not changed. The `if-else` has been replaced by a C conditional expression to save space; the form `a?b:c` means 'if `a` then `b` else `c`'.

For an example of statistics gathering, here is a program which constructs a histogram of the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
l s.
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement `return(1);` indicates that `lex` is to perform wrapup. If `yywrap` returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a `yywrap` that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double-precision FORTRAN to single-precision FORTRAN. Because FORTRAN does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes `double precision` to `real`, or `DOUBLE PRECISION` to `REAL`.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as 'beginning of line, then five blanks, then anything but blank or zero.' Note the two different meanings of `^`. There follow some rules to change double-precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+      |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p=+ 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D`. The program then adds `'e'-'d'`, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial `d`. By using the array `yytext` the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}     |
{d}{c}{o}{s}     |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext+1);
```

Another list of names must have initial d changed to initial a:

```
{d}{l}{o}{g}     |
{d}{l}{o}{g}10   |
{d}{m}{i}{n}1    |
{d}{m}{a}{x}1    {
        yytext[0] =+ 'a' - 'd';
        ECHO;
        }
```

And one routine must have initial d changed to initial r:

```
{d}1{m}{a}{c}{h}        {yytext[0] =+ 'r'  - 'd';
                ECHO;
                }
```

To avoid such names as dsinx being detected as instances of dsin, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*     |
[0-9]+   |
\n       |
.        ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

9.9. Left Context-Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the programmer's action code; such a flag is the simplest way of dealing with the problem, since lex is not involved at all. It may be more convenient, however, to have lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only be recognized when lex is in that start condition. The current start condition may be changed at any time. Finally,

if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word `magic` to `first` on every line which begins with the letter `a`, changing `magic` to `second` on every line which begins with the letter `b`, and changing `magic` to `third` on every line which begins with the letter `c`. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to `lex` in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word `Start` may be abbreviated to `s` or `S`. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when `lex` is in the start condition `name1`. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to `name1`. To resume the normal state,

```
BEGIN 0;
```

which resets to the initial condition of the `lex` automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but `lex` does the work rather than the programmer's code.

9.10. Character Set

The programs generated by `lex` handle character I/O only through the routines `input`, `output`, and `unput`. Thus the character representation provided in these routines is accepted by `lex` and employed to return values in `yytext`. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter `a` is represented in the same form as the character constant `'a'`.

If this interpretation is changed, by providing I/O routines which translate the characters, `lex` must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by two lines containing only '%T'. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example

Figure 9-3 *Sample character table.*

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear

either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

9.11. Summary of Source Format

The general form of a `lex` source file is:

```
{definitions}
%%
{rules}
%%
{programmer subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form 'name space translation'.
2. Included code, in the form 'space code'.
3. Included code, in the form

```
%{
code
%}
```

4. Start condition declarations, given in the form

```
%S name1 name2 ...
```

5. Character set tables, in the form

```
%T
number space character-string
...
%T
```

6. Changes to internal array sizes, in the form

   `%x nnn`

   where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Table 9-1 *Changing Internal Array Sizes in lex*

Letter	*Parameter*
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form 'expression action' where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in `lex` use the following operators:

Table 9-2 *Regular Expression Operators in lex*

Operator	*Meaning*
x	the character "x"
"x"	an "x", even if x is an operator
\x	an "x", even if x is an operator
[xy]	the character x or y
[x-z]	the characters x, y or z
[^x]	any character but x
.	any character but newline
^x	an x at the beginning of a line
<y>x	an x when lex is in start condition y
x$	an x at the end of a line
x?	an optional x
x*	0,1,2, ... instances of x
x+	1,2,3, ... instances of x
x|y	an x or a y
(x)	an x
x/y	an x but only if followed by y
{xx}	the translation of xx from the definitions section
x{m,n}	*m* through *n* occurrences of x

9.12. Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic automata; fortunately, they are rare.

`REJECT` does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and `REJECT` is executed, the programmer must not have used `unput` to change the characters forthcoming from the input stream. This is the only restriction on the programmer's ability to manipulate the not-yet-processed input.

10

yacc — Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an 'input language' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

yacc provides a general tool for describing the input to a computer program. The yacc programmer specifies the structure of the input, together with code to be invoked as each item is recognized. yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the programmer's application handled by this subroutine.

The input subroutine produced by yacc calls a programmer-supplied routine to return the next basic input item. Thus, the programmer can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The programmer-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

The class of specifications that yacc accepts is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, FORTRAN, APL, Pascal, Ratfor , etc., yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

yacc provides a general tool for imposing structure on the input to a computer program. The yacc programmer prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. yacc then generates a function to control the input process. This function, called a *parser*, calls the programmer-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then programmer code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

`yacc` generates its actions and output subroutines in C. Moreover, many of the syntactic conventions of `yacc` follow C.

The heart of the `yacc` input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
date  :  month_name  day  ',' year  ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma ',' is enclosed in single quotes — implying that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July  4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This routine reads the input stream, recognizing the lower-level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols are referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'  ;
month_name  :  'F' 'e' 'b'  ;
...
month_name  :  'D' 'e' 'c'  ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond `yacc`'s ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as ',' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be 'slipped in' to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe the basic process of preparing a yacc specification; Section 10.1 describes the preparation of grammar rules, Section 10.2 the preparation of the programmer-supplied actions associated with these rules, and Section 10.3 the preparation of lexical analyzers. Section 10.4 describes the operation of the parser. Section 10.5 discusses various reasons why yacc may be unable to produce a parser from a specification, and what to do about it. Section 10.6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 10.7 discusses error detection and recovery. Section 10.8 discusses the operating environment and special features of the parsers yacc produces. Section 10.9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10.10 discusses some advanced topics. Section 10.11 has a brief example, and section 10.12 gives a summary of the yacc input syntax. Section 10.13 gives an example using some of the more advanced features of yacc, and, finally, section 10.14 describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of yacc.

10.1. Basic Specifications

Names refer to either tokens or nonterminal symbols. `yacc` requires token names to be declared as such. In addition, for reasons discussed in Section 10.3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent `%%` marks. The percent `%` is generally used in `yacc` specifications as an escape character.

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second `%%` mark may be omitted also; thus, the smallest legal `yacc` specification is

```
%%
rules
```

Spaces (also called blanks), tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal — they are enclosed in `/* . . . */`, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are `yacc` punctuation.

Names may be of arbitrary length, and may be made up of letters, dot '.', underscore '_', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ''. As in C, the backslash '\' is an escape character within literals, and all the C escapes are recognized:

```
'\n'    newline
'\r'    return
'\''    single quote '
'\\'    backslash '\'
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  'xxx' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar '|' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D   ;
A       :       E  F   ;
A       :       G   ;
```

can be given to yacc as

```
A       :       B  C  D
        |       E  F
        |       G
        ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. See Sections 3 , 5, and 6 for much more discussion. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this

symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the programmer-supplied lexical analyzer to return the endmarker when appropriate — see Section 10.3, below. Usually the endmarker represents some reasonably obvious I/O status, such as 'end-of-file' or 'end-of-record'.

10.2. Actions

With each grammar rule, the programmer may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces '{' and '}'. For example,

```
A       :       '('  B  ')'
                        {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                        {       printf("a message\n");
                                flag = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign symbol '$' is used as a signal to `yacc` in this context.

To return a value, the action normally sets the pseudo-variable '$$' to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :       '('  expr  ')'          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of $1 (the first element in it). Thus, grammar rules of the form

```
A       :       B    ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                        {  $$ = 1;  }
                C
                        {   x = $2;   y = $3;  }
        ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. yacc actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                        {  $$ = 1;  }
        ;
A       :       B  $ACT  C
                        {   x = $2;   y = $3;  }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to

construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function `node`, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr    :       expr   '+'  expr
                         {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The programmer may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks '%{' and '%}'. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making `variable` accessible to all of the actions. The `yacc` parser uses only names beginning in 'yy'; the programmer should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.10.

10.3. Lexical Analysis

The programmer must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called `yylex()`. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable `yylval()`.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc`, or chosen by the programmer. In either case, the '# define' mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name `DIGIT` has been defined in the declarations section of the `yacc` specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
        ...
        case '0':
        case '1':
        ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return the token number of `DIGIT`, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier `DIGIT` will be defined as the token number associated with the token `DIGIT`.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of `if` or `while` as token names will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name `error` is reserved for error handling, and should not be used naively (see Section 10.7).

As mentioned above, the token numbers may be chosen by `yacc` or by the programmer. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the programmer; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *lex* program developed by Mike Lesk[8] and described in the previous chapter on `lex`. These lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications use regular expressions instead of grammar rules. `lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and

whose lexical analyzers must be crafted by hand.

10.4. How the Parser Works

`yacc` turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by `yacc` consists of a finite-state machine with a stack. The parser can read and remember the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite-state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls `yylex()` to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

`shift` Action

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is `IF`, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

`reduce` Action

The *reduce* action keeps the stack from growing without bound. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a '.') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
. reduce 18
```

refers to *grammar rule* 18, while the action

```
        IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       : x  y  z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
        A       goto 20
```

which pushes state 20 onto the stack, and becomes the current state.

In effect, the reduce action 'turns back the clock' in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of programmer-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable `yylval()` is copied onto the value stack. After the return from the programmer's code, the reduction is carried out. When the *goto* action is done, the external variable `yyval()` is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

`accept` and `error` Actions

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The

parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 10.7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When `yacc` is invoked with the **-v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, `DING`, is read, becoming the lookahead token. The action in state 0 on `DING` is 'shift 3', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, `DONG`, is read, becoming the lookahead token. The action in state 3 on the token `DONG` is 'shift 6', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3,

and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound  :  DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound  goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, DELL, must be read. The action is 'shift 5', so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by '$end' in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, and so on. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

10.5. Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr  :  expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not unambiguously specify the way that all complex inputs should be structured. For example, if the input is

```
expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
(  expr  -  expr  )  -  expr
```

or as

```
expr  -  (  expr  -  expr  )
```

The first is called *left association*, the second *right association*.

yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an

input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left-associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict.* It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict.* Note that there are never any 'shift/shift' conflicts.

When there are shift/reduce or reduce/reduce conflicts, `yacc` still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule.*

`yacc` invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the programmer rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than `yacc` can construct. The use of actions within rules can also cause conflicts, if the action must

be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, `yacc` always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, `yacc` will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an 'if-then-else' construction:

```
stat    :       IF  '('  cond  ')'  stat
        |       IF  '('  cond  ')'  stat  ELSE  stat
        ;
```

In these rules, `IF` and `ELSE` are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if rule*, and the second the *if-else rule*.

These two rules form an ambiguous construction, since input of the form:

```
IF ( condition-1 )  IF ( condition-2 ) statement-1  ELSE statement-2
```

can be structured according to these rules in two ways:

```
IF  (  condition-1  )  {
          IF  (  condition-2  )  statement-1
}
ELSE  statement-2
```

or

```
IF  (  condition-1  )  {
          IF  (  condition-2  )  statement-1
          ELSE  statement-2
}
```

The second interpretation is the one given in most programming languages having this construct. Each `ELSE` is associated with the last preceding 'un-*ELSE*'d' `IF`. In this example, consider the situation where the parser has seen

```
IF  (  condition-1  )  IF  (  condition-2  )  statement-1
```

and is looking at the `ELSE`. It can immediately reduce by the simple-if rule to get

```
IF  (  condition-1  )  stat
```

and then read the remaining input,

```
ELSE  statement-2
```

and reduce

```
IF  (  condition-1  )  stat  ELSE  statement-2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, *statement*-2 read, and then the right hand portion of

```
IF  (  condition-1  )  IF  (  condition-2  )  statement-1  ELSE  statement-2
```

can be reduced by the if-else rule to get

```
IF  (  condition-1  )  stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF  (  condition-1  )  IF  (  condition-2  )  statement-1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (–v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat  :  IF  (  cond  )  stat_          (18)
        stat  :  IF  (  cond  )  stat_ELSE  stat

      ELSE      shift 45
      .         reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is `ELSE`, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

since the `ELSE` will have been shifted in this state. Back in state 23, the alternative action, described by '.', is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not `ELSE`, the parser reduces by grammar rule 18:

```
stat  :  IF  '('  cond  ')'  stat
```

Once again, notice that the numbers following 'shift' commands refer to other states, while the numbers following 'reduce' commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most states, there will be at most one reduce action possible in the state, and this will be the default command. Programmers who encounter unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the programmer might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references cited in Chapter 1 might be consulted.

10.6. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr  :  expr  OP  expr
```

and

```
expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the programmer specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow `yacc` to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a `yacc` keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left   '+'   '-'
%left   '*'   '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left-associative, and have lower precedence than star and slash, which are also left-associative. The keyword `%right` is used to describe right-associative operators, and the keyword `%nonassoc` is used to describe operators, like the `.LT.` operator in FORTRAN, that may not associate with themselves; thus,

```
A  .LT.  B  .LT.  C
```

is illegal in FORTRAN, and such an operator would be described with the keyword `%nonassoc` in `yacc`. As an example of the behavior of these declarations, the description

```
%right   '='
%left   '+'   '-'
%left   '*'   '/'

%%

expr    :       expr   '='  expr
        |       expr   '+'  expr
        |       expr   '-'  expr
        |       expr   '*'  expr
        |       expr   '/'  expr
        |       NAME
        ;
```

might be used to structure the input

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword `%prec` changes the precedence level associated with a particular grammar rule. `%prec` appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It changes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left   '+'   '-'
%left   '*'   '/'

%%

expr    :       expr   '+'  expr
        |       expr   '-'  expr
        |       expr   '*'  expr
        |       expr   '/'  expr
        |       '-'  expr       %prec   '*'
        |       NAME
        ;
```

A token declared by `%left`, `%right`, and `%nonassoc` need not be, but may be, declared by `%token` as well.

The precedences and associativities are used by `yacc` to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left-associative implies reduce, right-associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by `yacc`. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially 'cookbook' fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

10.7. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser 'restarted' after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the programmer some control over this process, `yacc` provides a simple, but reasonably general, feature. The token name 'error' is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token 'error' is legal. It then behaves as if the token 'error' were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :       error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any 'cleanup' action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error   '\n'  {  printf( "Reenter last line: " );  }  input
                        {       $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error   '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " );    }
                input
                        {       $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the 'error' symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the programmer, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by `yylex()` would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;    }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the programmer can get control to deal with the error actions required by other portions of the program.

10.8. The yacc Environment

When the programmer inputs a specification to yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the name may differ from installation to installation). yacc produces an integer-valued function called yyparse(). When yyparse() is called, it in turn repeatedly calls yylex() — the lexical analyzer supplied by the programmer (see Section 10.3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse() returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, yyparse() returns the value 0.

The programmer must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called main must be defined, that eventually calls yyparse(). In addition, a routine called yyerror() prints a message when a syntax error is detected.

The programmer must supply these two routines in one form or another. They can be as simple as the following example, or they can be as complex as needed.

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to yyerror() is a string containing an error message, usually the string 'syntax error'. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser generates a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

10.9. Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of 'knowing who to blame when things go wrong.'
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be added easily.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in section 10.11 is written following this style, as are the examples in the text of this paper (where space permits). The programmer must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the `yacc` parser encourages so called 'left-recursive' grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right-recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the programmer should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
        other declarations
%%
prog    :       decls  stats
        ;
decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;
stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;
        other rules
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except* for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single-token exception does not affect the lexical scan.

This kind of 'backdoor' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the programmer to use words like 'if', which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of `yacc`; it is difficult to pass information to the lexical analyzer telling it 'this instance of `if` is a keyword, and that instance is a variable'. The programmer can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10.10. Advanced Topics

This section discusses a number of advanced features of `yacc`.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT makes `yyparse` return the value 0; YYERROR makes the parser behave as if the current input symbol results in a syntax error; `yyerror()` is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :   adj  noun  verb  adj  noun
            {  look at the sentence . . .  }
    ;

adj :   THE     {   $$ = THE;   }
    |   YOUNG   {   $$ = YOUNG;  }
    . . .
    ;

noun    :   DOG
            {   $$ = DOG;   }
    |   CRONE
            {   if( $0 == YOUNG ){
                    printf( "what?\n" );
                    }
                $$ = CRONE;
                }
    ;
    . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. yacc can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack (see Section 10.4) is declared to be a union of the various types of values desired. The programmer declares the union, and associates a union member name to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc automatically inserts the appropriate union name, so that no unwanted conversions will take place. In addition, type-checking commands such as lint(1) will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the programmer since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where yacc cannot easily determine the type.

To declare the union, the programmer includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the yacc value stack, and the external variables yylval and yyval, to have type equal to this union. If yacc was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  '+'  '-'
```

will tag any reference to values returned by these two tokens with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left-context values (such as $0 — see the previous subsection) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :   aaa  {  $<intval>$  =  3;  } bbb
            {   fun( $<intval>2, $<other>0 );  }
    ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in 10.13. The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the `yacc` value stack is used to hold `int`'s, as was true historically. This paper is reprinted in this manual.

10.11. A Simple Example

This example gives the complete `yacc` specification for a small desk calculator; the desk calculator has 26 registers, labeled 'a' through 'z', and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a `yacc` specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line-by-line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start  list

%token  DIGIT  LETTER

%left  '|'
%left  '&'
```

```
%left  '+'  '-'
%left  '*'  '/'  '%'
%left UMINUS   /* supplies precedence for unary minus */

%%   /* beginning of rules section */

list    :       /* empty */
        |       list  stat  '\n'
        |       list  error  '\n'
                        {       yyerrok;  }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 );  }
        |       LETTER  '='  expr
                        {       regs[$1]  =  $3;  }
        ;

expr    :       '('  expr  ')'
                        {       $$  =  $2;  }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  '-'  expr
                        {       $$  =  $1  -  $3;  }
        |       expr  '*'  expr
                        {       $$  =  $1  *  $3;  }
        |       expr  '/'  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  '%'  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  '&'  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  '|'  expr
                        {       $$  =  $1  |  $3;  }
        |       '-'  expr        %prec  UMINUS
                        {       $$  =  -  $2;  }
        |       LETTER
                        {       $$  =  regs[$1];  }
        |       number
        ;

number  :       DIGIT
                        {       $$ = $1;    base  =  ($1==0)  ?  8  :  10;  }
        |       number  DIGIT
                        {       $$  =  base * $1  +  $2;  }
        ;

%%   /* start of programs */

yylex()         /* lexical analysis routine */
{
/* returns LETTER for lower case letter, yylval=0 thru 25 */
/* return DIGIT for digit, yylval=0 thru 9 */
/* all other characters are returned immediately */

        int     c;

        while((c = getchar()) == ' ') { /* skip blanks */ }
```

```
                                        /* c is now nonblank */
        if(islower(c))  {
                yylval = c - 'a';
                return(LETTER);
        }
        if(isdigit(c))  {
                yylval = c - '0';
                return(DIGIT);
        }
        return(c);
}
```

10.12. yacc Input Syntax

This section describes the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (not literal) followed by : */
%token  NUMBER          /* [0-9]+ */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs  MARK  rules  tail
        ;

tail    :       MARK    {    In this action, eat up the rest of the file    }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs  def
        ;
```

```
def     :       START  IDENTIFIER
        |       UNION  {  Copy union definition to output  }
        |       LCURL  {  Copy C code to output file  }  RCURL
        |       defs  rword  tag  nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<'  IDENTIFIER  '>'
        ;

nlist   :       nmno
        |       nlist  nmno
        |       nlist  ','  nmno
        ;

nmno    :       IDENTIFIER      /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER   /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER  rbody  prec
        |       rules  rule
        ;

rule    :       C_IDENTIFIER  rbody  prec
        |       '|'  rbody  prec
        ;

rbody   :       /* empty */
        |       rbody  IDENTIFIER
        |       rbody  act
        ;

act     :       '{'  {  Copy action, translate $$, etc.  }  '}'
        ;

prec    :       /* empty */
        |       PREC  IDENTIFIER
        |       PREC  IDENTIFIER  act
        |       prec  ';'
        ;
```

10.13. An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in Section 10.10. The desk calculator example in section 10.11 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, –, *, /, unary –, and = (assignment), and has 26 floating point variables, 'a' through 'z'. Moreover, it also understands *intervals*, written

```
( x , y )
```

where x is less than or equal to y. There are 26 interval-valued variables 'A' through 'Z' that may also be used. The usage is similar to that in section 10.11 — assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of `yacc` and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, `INTERVAL`, by using `typedef`.
The `yacc` value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of `YYERROR` to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of `yacc` is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval-value. This causes a large number of conflicts when the grammar is run through `yacc`: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval-valued expression in the second example, but this fact is not known until the ',' is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval-valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar-valued expressions scalar-valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more

normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>
typedef  struct  interval  {
         double  lo,  hi;
         }  INTERVAL;
INTERVAL  vmul(),  vdiv();
double  atof();
double  dreg[ 26 ];
INTERVAL  vreg[ 26 ];
%}
%start    lines
%union    {
         int  ival;
         double  dval;
         INTERVAL  vval;
         }
%token <ival> DREG VREG    /* indices into dreg, vreg arrays */
%token <dval> CONST                /* floating point constant */
%type <dval> dexp          /* expression */
%type <vval> vexp          /* interval expression */
         /* precedence information about the operators */
%left    '+'  '-'
%left    '*'  '/'
%left    UMINUS    /* precedence for unary minus */
%%
lines    :        /* empty */
         |        lines  line
         ;
line     :        dexp  '\n'
                           {        printf(  "%15.8f\n",  $1  );  }
         |        vexp  '\n'
                           {        printf(  "(%15.8f  ,  %15.8f  )\n",  $1.lo,  $1.hi  );  }
         |        DREG  '='  dexp  '\n'
                           {        dreg[$1]  =  $3;  }
         |        VREG  '='  vexp  '\n'
                           {        vreg[$1]  =  $3;  }
         |        error  '\n'
                           {        yyerrok;  }
```

```
        ;

dexp    :       CONST
        |       DREG
                        {       $$ = dreg[$1]; }
        |       dexp '+' dexp
                        {       $$ = $1 + $3; }
        |       dexp '-' dexp
                        {       $$ = $1 - $3; }
        |       dexp '*' dexp
                        {       $$ = $1 * $3; }
        |       dexp '/' dexp
                        {       $$ = $1 / $3; }
        |       '-' dexp        %prec UMINUS
                        {       $$ = - $2; }
        |       '(' dexp ')'
                        {       $$ = $2; }
        ;

vexp    :       dexp
                        {       $$.hi = $$.lo = $1; }
        |       '(' dexp ',' dexp ')'
                        {
                        $$.lo = $2;
                        $$.hi = $4;
                        if( $$.lo > $$.hi ){
                                printf( "interval out of order\n" );
                                YYERROR;
                                }
                        }
        |       VREG
                        {       $$ = vreg[$1];  }
        |       vexp '+' vexp
                        {       $$.hi = $1.hi + $3.hi;
                                $$.lo = $1.lo + $3.lo;  }
        |       dexp '+' vexp
                        {       $$.hi = $1 + $3.hi;
                                $$.lo = $1 + $3.lo;  }
        |       vexp '-' vexp
                        {       $$.hi = $1.hi - $3.lo;
                                $$.lo = $1.lo - $3.hi;  }
        |       dexp '-' vexp
                        {       $$.hi = $1 - $3.lo;
                                $$.lo = $1 - $3.hi;  }
        |       vexp '*' vexp
                        {       $$ = vmul( $1.lo, $1.hi, $3 ); }
        |       dexp '*' vexp
                        {       $$ = vmul( $1, $1, $3 ); }
        |       vexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1.lo, $1.hi, $3 ); }
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = -$2.lo;  $$.lo = -$2.hi;  }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;
```

```
%%
# define BSZ 50  /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register  c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if(  isupper(  c  )  ){
                yylval.ival  =  c  -  'A';
                return(  VREG  );
                }
        if(  islower(  c  )  ){
                yylval.ival  =  c  -  'a';
                return(  DREG  );
                }

        if(  isdigit(  c  )  ||  c=='.'  ){
                /* gobble up digits, points, exponents */

                char  buf[BSZ+1],  *cp  =  buf;
                int  dot  =  0,  exp  =  0;

                for(  ;  (cp-buf)<BSZ  ;  ++cp,c=getchar()  ){

                        *cp  =  c;
                        if(  isdigit(  c  )  )  continue;
                        if(  c  ==  '.'  ){
                                if( dot++ || exp ) return( '.' );
                                        /* will cause syntax error */
                                continue;
                                }

                        if(  c  ==  'e'  ){
                                if( exp++ ) return( 'e' );
                                        /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp  =  '\0';
                if(  (cp-buf)  >=  BSZ  )
                        printf(  "constant  too  long:  truncated\n"  );
                else ungetc( c, stdin );  /* push back last char read */
                yylval.dval  =  atof(  buf  );
                return(  CONST  );
                }
        return(  c  );
        }

INTERVAL  hilo(  a,  b,  c,  d  )  double  a,  b,  c,  d;  {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL  v;

        if(  a>b  )  {  v.hi  =  a;    v.lo  =  b;  }
        else  {  v.hi  =  b;    v.lo  =  a;  }

        if(  c>d  )  {
                if(  c>v.hi  )  v.hi  =  c;
```

```
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

10.14. Old Features Supported but not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes '"'.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with `yacc`, since it suggests that `yacc` is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash '\' may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

```
%< is the same as %left
%> is the same as %right
%binary and %2 are the same as %nonassoc
%0 and %term are the same as %token
%= is the same as %prec
```

5. Actions may also have the form

   ```
   ={ . . . }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

11

The curses Library: Screen-Oriented Cursor Motions

curses is a Library Package for:

- Updating a screen with reasonable optimization,
- Getting input from the terminal in a screen-oriented fashion, and
- Moving the cursor from one point to another, independent of the two previous functions.

These routines all use the termcap database to describe the capabilities of the terminal.

Overview

In making available the generalized terminal descriptions in termcap, much information was made available to the programmer, but little work was taken out of one's hands. curses helps the programmer perform the required functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The curses package is split into three parts:

1. Screen updating without user input;
2. Screen updating with user input; and
3. Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the termcap database itself.

Terminology

In this chapter, the terminology illustrated in the table below is used with reasonable consistency.

Table 11-1 *Description of Terms*

Term	*Description*
window	An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen. Note that the term *window* is used elsewhere in the Sun system manuals when describing the window management packages for driving the bitmapped screens. `curses` windows bear little, if any, resemblance to the window system concepts.
terminal	Sometimes called *terminal screen*. The package's idea of what the terminal's screen currently looks like, that is, what the user sees now. This is a special screen:
screen	This is a subset of windows which are as large as the terminal screen, that is, they start at the upper left hand corner and encompass the lower right hand corner. One of these, `stdscr`, is automatically provided for the programmer.

Cursor Addressing Conventions

The `curses` library routines address positions on a screen with the `y` coordinate first and the `x` coordinate second. This follows the convention of most terminals that address the screen in `row`, `column` order. The reader should note this convention.

Compiling Things

To use the `curses` library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

```
#include <curses.h>
```

at the top of the program source.[29]

Also, compilations should have the following form:

```
tutorial% cc [ C-compiler options ] filename ... -lcurses -ltermcap
```

[29] The header file `<curses.h>` needs to include `<sgtty.h>`, so one should not do so oneself. The screen package also uses the Standard I/O library, so `<curses.h>` includes `<stdio.h>`. It is redundant (but harmless) to include it again.

Screen Updating

To update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named `window()` is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) coordinates of the upper left hand corner) and its size. One of these (called `curscr` for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called `stdscr`, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine `refresh()` (or `wrefresh()` if the window is not `stdscr`) is called. `refresh()` makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal.* Actual updates to the terminal screen are made only by calling `refresh()` or `wrefresh()`. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say 'make it look like this,' and let the package worry about the best way to do this.

Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: `curscr`, which knows what the terminal looks like, and `stdscr`, which is what the programmer wants the terminal to look like next. The user should never really access `curscr` directly. Changes should be made to the appropriate screen, and then the routine `refresh()` (or `wrefresh()`) should be called.

Many functions are set up to deal with `stdscr` as a default screen. For example, to add a character to `stdscr`, one calls `addch()` with the desired character. If a different window is to be used, the routine `waddch()` (for "window-specific" `addch()`) is provided[30]. This convention of prepending function names with a `w` when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

[30] Actually, `addch()` is really a macro with arguments, as are most of the "functions" which deal with `stdscr` as a default.

To move the current (y, x) coordinates from one point to another, the routines `move()` and `wmove()` are provided. However, it is often desirable to first move and then perform some I/O operation. To avoid clumsiness, most I/O routines can be preceded by the prefix `mv` and the desired (y, x) coordinates then can be added to the arguments to the function. For example, the calls:

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (`win`) comes before the added (y, x) coordinates. If such pointers are needed, they are always the first parameters passed.

11.1. Variables

Many variables that describe the terminal environment are available to the programmer. They are:

Table 11-2 *Variables to Describe the Terminal Environment*

Type	*Name*	*Description*
`WINDOW *`	`curscr`	current version of the screen (terminal screen).
`WINDOW *`	`stdscr`	standard screen. Most updates are done here.
`char *`	`Def_term`	default terminal type if type cannot be determined
`bool`	`My_term`	use the terminal specification in `Def_term` as terminal, irrelevant of real terminal type
`char *`	`ttytype`	full name of the current terminal.
`int`	`LINES`	number of lines on the terminal
`int`	`COLS`	number of columns on the terminal
`int`	`ERR`	error flag returned by routines on a fail.
`int`	`OK`	error flag returned by routines when things go right.

There are also several #define constants and types which are of general usefulness:

reg	storage class register (for example, reg int i;)
bool	boolean type, actually a char (for example, bool doneit;)
TRUE	boolean 'true' flag (1).
FALSE	boolean 'false' flag (0).

11.2. Programming Curses

This is a description of how to actually use the screen package. In it, we assume all updating, reading, and so on, is applied to stdscr. All instructions will work on any window, by changing the function name and parameters as mentioned above.

Starting Up

To use the screen package, the routines must know about terminal characteristics, and the space for curscr and stdscr must be allocated. These functions are performed by initscr(). Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, initscr() returns ERR. initscr() must *always* be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either curscr or stdscr are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like nl() and cbreak() should be called after initscr().

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use scrollok(). If you want the cursor to be left after the last change, use leaveok(). If this isn't done, refresh() moves the cursor to the window's current (y, x) coordinates after updating it. New windows of your own can be created, too, by using the functions newwin() and subwin(). delwin() gets rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables LINES and COLS to be what you want, and then call initscr(). This is best done before, but can be done either before or after, the first call to initscr(), as it always deletes any existing stdscr and/or curscr before creating new ones.

The Nitty-Gritty

Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what appears on a window are addch() and move(). addch() adds a character at the current (y, x) coordinates, returning ERR if it would cause the window to illegally scroll, that is, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. move() changes the current (y, x) coordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into mvaddch() to do both things in one fell swoop.

The other output functions, such as `addstr()` and `printw()`, all call `addch()` to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call `refresh()`. To optimize finding changes, `refresh()` assumes that any part of the window not changed since the last `refresh()` of that window has not been changed on the terminal, that is, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routines `touchwin()`, `touchline()`, and `touchoverlap()` are provided to make it look like the entire window has been changed, thus forcing `refresh()` check the whole subsection of the terminal for changes.

If you call `wrefresh()` with `curscr`, it will make the screen look like `curscr` thinks it looks like. This is useful for implementing a command to redraw the screen in case it get messed up.

Input

Input is essentially a mirror image of output. The complementary function to `addch()` is `getch()` which, if echo is set, calls `addch()` to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in *raw* or *cbreak* mode. If it is not, `getch()` sets it to be cbreak, reads in the character, and then resets the mode of the terminal to what it was before the call.

Miscellaneous

All sorts of functions exist for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

Finishing Up

To do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in `getttmode()` and `setterm()`, which are called by `initscr()`. To clean up after the routines, the routine `endwin()` is provided. It restores tty modes to what they were when `initscr()` was first called. Thus, anytime after the call to initscr, `endwin()` should be called before exiting.

11.3. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "crt hacks"[31] and optimizing `cat`(1)-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

[31] Graphics programs designed to run on character-oriented terminals.

Terminal Information

To use a terminal's features to the best of a program's abilities, you must first know what they are. The termcap database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that curses uses is taken from vi(1) and is efficient. It reads them into a set of variables whose names are two uppercase letters with some mnemonic value. For example, HO is a string which moves the cursor to the "home" position[32]. As there are two types of variables involving ttys, there are two routines. The first, gettmode(), sets some variables based upon the tty modes accessed by gtty(2) and stty(2). The second, setterm(), does a larger task by reading in the descriptions from the termcap database. This is the way these routines are used by initscr():

```
if (isatty(0)) {
        gettmode;
        if (sp=getenv("TERM"))
                setterm(sp);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal[33]. If it is, gettmode() sets the terminal description modes from a gtty(2). getenv() is then called to get the name of the terminal, and that value (if there is one) is passed to setterm(), which reads in the variables from termcap associated with that terminal. getenv() returns a pointer to a string containing the name of the terminal, which we save in the character pointer sp. If isatty() returns false, the default terminal Def_term is used. The TI and VS sequences initialize the terminal. _puts() is a macro which uses tputs() (see termcap(3X)) to put out a string. It is these things which endwin() undoes.

Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task.

After using gettmode() and setterm() to get the terminal descriptions, the function mvcur() deals with this task. Its usage is simple: you simply tell it where you are now and where you want to go, as shown below.

---

[32] These names are identical to those variables used in the /etc/termcap database to describe each capability. See Appendix A for a complete list of those read, and termcap(5) for a full description.

[33] isatty() is defined in the default C library function routines. It does a gtty(2) on the file descriptor and checks the return value.

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function `tgoto()` from the `termcap`(3X) routines, or you can tell `mvcur()` that you are impossibly far away. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

11.4. Curses Functions

In the following definitions, '' means that the 'function' is really a `#define` macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as `addch()`, it will show up as its 'w' counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

Output Functions

`addch()` and `waddch()` — Add Character to Window

```
addch(ch)
char    ch;

waddch(win, ch)
WINDOW  *win;
char    ch;
```

Add the character `ch` on the window at the current (y, x) co-ordinates. If the character is a NEWLINE (`\n`) the line is cleared to the end, and the current (y, x) co-ordinates are changed to the beginning of the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return (`\r`) moves to the beginning of the line on the window. Tabs (`\t`) are expanded into spaces in the normal tabstop positions of every eight characters. This returns `ERR` if it would cause the screen to scroll illegally.

`addstr()` and `waddstr()` — Add String to Window

```
addstr(st)
char    *str;

waddstr(win, str)
WINDOW  *win;
char    *str;
```

Add the string pointed to by `str` on the window at the current (y, x) co-ordinates. This returns `ERR` if it would cause the screen to scroll illegally. In this case, it puts on as much as it can.

box() — Draw Box Around Window

```
box(win, vert, hor)
WINDOW  *win;
char    vert, hor;
```

Draws a box around the window using `vert` as the character for drawing the vertical sides, and `hor` for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear() and wclear() — Reset Window

```
clear()

wclear(win)
WINDOW  *win;
```

Resets the entire window to blanks. If `win` is a screen, this sets the clear flag, which sends a clear-screen sequence on the next `refresh()` call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok() — Set Clear Flag

```
clearok(scr, boolf)
WINDOW  *scr;
bool    boolf;
```

Sets the clear flag for the screen `scr`. If `boolf` is `TRUE`, this forces a clear-screen to be printed on the next `refresh()`, or stop it from doing so if `boolf` is `FALSE`. This only works on screens, and, unlike `clear()`, does not alter the contents of the screen. If `scr` is `curscr`, the next `refresh()` call causes a clear-screen, even if the window passed to `refresh()` is not a screen.

clrtobot() and wclrtobot() — Clear to Bottom

```
clrtobot()

wclrtobot(win)
WINDOW  *win;
```

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated `mv` function.

clrtoeol() and wclrtoeol() — Clear to End of Line

```
clrtoeol()

wclrtoeol(win)
WINDOW  *win;
```

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated `mv` function.

delch() and wdelch() — Delete Character

```
delch()

wdelch(win)
WINDOW  *win;
```

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln() and wdeleteln() — Delete Current Line

```
deleteln()

wdeleteln(win)
WINDOW  *win;
```

Delete the current line. Every line below the current one moves up, and the bottom line becomes blank. The current (y, x) co-ordinates remains unchanged.

erase and werase() — Erase Window

```
erase()

werase(win)
WINDOW  *win;
```

Erases the window to blanks without setting the clear flag. This is analagous to `clear()`, except that it never causes a clear-screen sequence to be generated on a `refresh()`. This has no associated mv function.

flushok — Control Flushing of stdout

```
flushok(win, boolf)
WINDOW  *win;
bool boolf;
```

Normally, `refresh()` performs an `fflush()` on `stdout` when it is finished. `flushok()` allows you to control this. If `boolf` is TRUE (non-zero), `refresh()` performs the `fflush()`; if FALSE, `refresh()` does not.

idlok — Control Use of Insert/Delete Line

```
idlok(win, boolf)
WINDOW  *win;
bool boolf;
```

Reserved for future use. When implemented, this will signal `refresh()` as to whether it is safe to use "insert line" and "delete line" sequences to update a window.

insch() and winsch() — Insert Character

```
insch(c)
char    c;

winsch(win, c)
WINDOW  *win;
char    c;
```

Insert `c` at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns `ERR` if it would cause the screen to scroll illegally.

insertln() and winsertln() — Insert Line

```
insertln

winsertln(win)
WINDOW  *win;
```

Insert a line above the current one. Every line below the current line is shifted down, and the bottom line disappears. The current line becomes blank, and the current (y, x) co-ordinates remains unchanged. This returns ERR if it would cause the screen to scroll illegally.

move and wmove() — Move

```
move(y, x)
int y, x;

wmove(win, y, x)
WINDOW  *win;
int y, x;
```

Change the current (y, x) co-ordinates of the window to y, x. This returns ERR if it would cause the screen to scroll illegally.

overlay() — Overlay Windows

```
overlay(win1, win2)
WINDOW  *win1, *win2;
```

Overlay win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done non-destructively, that is, blanks on win1 leave the contents of the space on win2 untouched.

overwrite() — Overwrite Windows

```
overwrite(win1, win2)
WINDOW  *win1, *win2;
```

Overwrite win1 on win2. The contents of win1, insofar as they fit, are placed on win2 at their starting (y, x) co-ordinates. This is done destructively, that is, blanks on win1 become blank on win2.

printw() and wprintw() — Print to Window

```
printw(fmt, arg1, arg2, ...)
char    *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW  *win;
char    *fmt;
```

Performs a printf() on the window starting at the current (y, x) co-ordinates. It uses addstr() to add the string on the window. It is often advisable to use the field width options of printf() to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh() and wrefresh() — Synchronize

```
refresh()

wrefresh(win)
WINDOW  *win;
```

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it updates whatever it can without causing the scroll.

As a special case, if `wrefresh()` is called with the window `curscr`, the screen is cleared and repainted. This is useful for allowing the user to redraw the screen as needed.

standout() and wstandout() — Put Characters in Standout Mode

```
standout()

wstandout(win)
WINDOW  *win;

standend()

wstandend(win)
WINDOW  *win;
```

Start and stop putting characters onto `win` in standout() mode. `standout()` causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). `standend()` stops this. The sequences `SO` and `SE` (or `US` and `UE` if they are not defined) are used (see Appendix A).

Input Functions

crbreak and nocrbreak — Set or Unset from Cbreak mode

```
crbreak()

nocrbreak()
```

Set or unset the terminal to/from cbreak mode. The misnamed macros `crmode()` and `nocrmode()` are retained for backward compatibility.

echo() and noecho() — Turn Echo On or Off

```
echo()

noecho()
```

Sets the terminal to echo or not echo characters.

getch() and wgetch() — Get Character from Terminal

```
getch()

wgetch(win)
WINDOW  *win;
```

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If `noecho()` has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of

noecho(), cbreak(), or rawmode set. If you do not set one, whatever routine you call to read characters sets cbreak for you, and then resets to the original mode when finished.

getstr() and wgetstr() — Get String from Terminal

```
getstr(st)
char    *str;

wgetstr(win, str)
WINDOW  *win;
char    *str;
```

Get a string through the window and put it in the location pointed to by str, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls getch() (or wgetch(win)) to get the characters needed to fill in the string until a NEWLINE or EOF is encountered. The NEWLINE is stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw() and noraw() — Turn Raw Mode On or Off

```
raw()

noraw()
```

Set or unset the terminal to/from raw mode. On version 7 UNIX† systems, this also turns off NEWLINE mapping (see nl()).

scanw() and wscanw() — Read String from Terminal

```
scanw(fmt, arg1, arg2, ...)
char    *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW  *win;
char    *fmt;
```

Perform a scanf() through the window using fmt. It does this using consecutive getch()'s (or wgetch(win)'s). This returns ERR if it would cause the screen to scroll illegally.

Miscellaneous Functions

baudrate — Get the Baudrate

Returns the baud rate of the terminal. This is a system-dependent constant (defined in the header file <sys/tty.h>, which is included in <curses.h>).

† UNIX is a registered trademark of AT&T.

`delwin()` — Delete a Window

```
delwin(win)
WINDOW  *win;
```

Deletes the window from existence. All resources are freed for future use by `calloc(3)`. If a window has a `subwin()` allocated window inside of it, deleting the outer window does not affect the subwindow, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

`endwin()` — Finish up Window Routines

```
endwin()
```

Finish up window routines before exit. This restores the terminal to the state it was in before `initscr()` (or `gettmode()` and `setterm()`) was called. `endwin()` should always be called before exiting. `endwin()` does not itself exit — this is especially useful for resetting tty stats when trapping rubouts via `signal(2)`.

`erasechar` — Get Erase Character

```
erasechar()
```

Returns the erase character for the terminal; that is, the character used by the terminal to erase single characters from the input.

`getcap()` — Get Termcap Capability

```
char *getcap(str)
char *str;
```

Return a pointer th the `termcap` capability described by `str` (see `termcap`(5) for details).

`getyx()` — Get Current Coordinates

```
getyx(win, y, x)
WINDOW  *win;
int y, x;
```

Puts the current (y, x) co-ordinates of `win` in the variables `y` and `x`. Since it is a macro, not a function, you do not pass the address of `y` and `x`.

`inch()` and `winch()` — Get Character at Current Coordinates

```
inch()
```

```
winch(win)
WINDOW  *win;
```

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated `mv` function.

`initscr()` — Initialize Screen Routines

```
initscr()
```

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by `Def_term` (initialy `dumb`). If the

boolean My_term is true, Def_term is always used. If the window size values for rows and columns as returned by the TIOCGWINSZ ioctl(2) request are non-zero, they are used. Otherwise, sizes are taken from the termcap description.

killchar — Get Kill Character

```
killchar()
```

Returns the terminal's line kill character; that is, the character used to erase an entire line from input.

leaveok() — Set Leave Cursor Flag

```
leaveok(win, boolf)
WINDOW  *win;
bool    boolf;
```

Sets the boolean flag for leaving the cursor after the last change. If boolf is TRUE, the cursor is left after the last update on the terminal, and the current (y, x) co-ordinates for win are changed accordingly. If it is FALSE, it is moved to the current (y, x) co-ordinates. This flag (initially FALSE) retains its value until changed by the user.

For example, say the current position is (0, 0) and we change the character at position (5, 10) in the window. After calling refresh(), the cursor is either moved to position (5, 10) (if the flag is TRUE) or the cursor is left at position (0, 0) (if the flag is FALSE).

longname() — Get Full Name of Terminal

```
longname(termbuf, name)
char    *termbuf, *name;

longname(termbuf, name)
char    *termbuf, *name;
```

Fills in name with the long (full) name of the terminal described by the termcap entry in termbuf. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable ttytype. termbuf is usually set via the termcap routine tgetent. fullname is the same as longname(), except that it gives the fullest name given in the entry, which can be quite verbose.

mvwin — Move Home Position of Window

```
mvwin(win, y, x)
WINDOW  *win;
int y, x;
```

Move the home position of the window win from its current starting coordinates to y, x. If that would put part or all of the window off the edge of the terminal screen, mvwin() returns ERR and does not change anything. For subwindows, mvwin() also returns ERR if you attempt to move it off its main window. If you move a main window, all subwindows are moved along with it.

newwin() — Create a New Window

```
WINDOW *
newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;
```

Create a new window with `lines` lines and `cols` columns starting at position `begin_y, begin_x`. If either `lines` or `cols` is 0 (zero), that dimension is set to (`lines - begin_y`) or (`cols - begin_x`) respectively. Thus, to get a new window of dimensions `lines` × `cols`, use `newwin(0, 0, 0, 0)`.

nl() and nonl() — Turn Newline Mode On or Off

```
nl()
nonl()
```

Set or unset the terminal to/from `nl()` mode, that is, start/stop the system from mapping [RETURN] to [NEWLINE]. If the mapping is not done, `refresh()` can do more optimization, so it is recommended, but not required, that it be turned off.

scrollok — Set Scroll Flag for Window

```
scrollok(win, boolf)
WINDOW  *win;
bool    boolf;
```

Set the scroll flag for the given window. If `boolf` is `FALSE`, scrolling is not allowed. This is its default setting.

subwin() — Create a Subwindow

```
WINDOW *
subwin(win, lines, cols, begin_y, begin_x)
WINDOW  *win;
int lines, cols, begin_y, begin_x;
```

Create a new window with `lines` lines and `cols` columns starting at position (`begin_y, begin_x`) in the middle of the window `win`. This means that any change made to either window in the area covered by the subwindow is made on both windows. (`begin_y, begin_x`) are specified relative to the overall screen, not the relative (0, 0) of `win`. If either `lines` or `cols` is 0 (zero), that dimension is set to (`LINES - begin_y`) or (`COLS - begin_x`) respectively.

touchline — Indicate Line Has Been Changed

```
touchline(win, y, startx, endx)
WINDOW  *win;
int y, startx, endx;
```

This function performs a function similar to `touchwin()`, but on a single line. It marks the first change for the given line to be `startx`, if it is before the current first change mark, and the last change mark is set to be `endx` if it is currently less than `endx`.

`touchoverlap` — Indicate Overlapping Regions Have Been Changed

```
touchoverlap(win1, win2)
WINDOW  *win, *win2;
```

Touch the window `win2` in the area which overlaps with `win1`. If they do not overlap, no changes are made.

`touchwin()` — Indicate Window Has Been Changed

```
touchwin(win)
WINDOW  *win;
```

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

`unctrl()` — Return Representation of Character

```
unctrl(ch)
char    ch;
```

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of `ch`. Control characters become their upper-case equivalents preceded by a ^ (circumflex character). Other letters stay just as they are.

Details

`gettmode()` — Get tty Statistics

```
gettmode()
```

Get the tty stats. This is normally called by `initscr()`.

`mvcur()` — Move Cursor

```
mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;
```

Moves the terminal's cursor from `lasty`, `lastx` to `newy`, `newx` in an approximation of optimal fashion.

It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. `move()` and `refresh()` should be used to move the cursor position, so that the routines know what's going on.

`scroll()` — Scroll Window

```
scroll(win)
WINDOW  *win;
```

Scroll the window upward one line. This is normally not used by the user.

`savetty()` and `resetty()` — Save and Reset tty Flags

```
savetty()

resetty()
```

`savetty()` saves the current tty characteristic flags. `resetty()` restores them to what `savetty()` stored. These functions are performed automatically by `initscr()` and `endwin()`.

`setterm()` — Set Terminal Characteristics

```
setterm(name)
char    *name;
```

Set the terminal characteristics to be those of the terminal named `name`, getting the terminal size from the `TIOCGWINSZ ioctl(2)` request if that size is non-zero, and otherwise from the environment. This is normally called by `initscr()`.

`tstp`

```
tstp()
```

This function saves the current tty state and then puts the process to sleep. When the process gets restarted, it restores the tty state and then calls `wrefresh(curscr)` to redraw the screen. The `initscr()` function sets the signal `SIGTSTP` to trap to this routine.

`_putchar()`

```
_putchar()
```

Put out a character using the `putchar()` macro. This function is used to output every character that `curses` generates. Thus, it can be redefined by the user who wants to do non-standard things with the output. It is named with an initial '_' because it usually should be invisible to the programmer.

11.5. Capabilities from `termcap`

Note that the description of terminals is a difficult business, and we only attempt to summarize the capabilities here. For a full description see the `termcap`(5) manual pages.

Overview

Capabilities from `termcap` are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a `LP` at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed — specified by `PC`). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, for example, `12*` before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say '`P*`'.

Variables Set By `setterm()`

Table 11-3 *Variables Set by* `setterm()`

Type	*Name*	*Pad*	*Description*
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with X are reserved for severely nauseous glitches

For purposes of `standout()`, if SG is not 0, SO is set to NULL, and if UG is not 0, US is set to NULL. If, after this, SO is NULL, and US is not, SO is set to be US, and SE is set to be UE.

Variables Set By `gettmode()`

Table 11-4 *Variables Set By* `gettmode()`

type	*name*	*description*
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

11.6. The `WINDOW` structure

The WINDOW structure is defined as follows:

```
/*
 * Copyright (c) 1980 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 *
 *        @(#)win_st.c        6.1 (Berkeley) 4/24/86";
 */

# define  WINDOW    struct _win_st

struct _win_st {
          short               _cury, _curx;
          short               _maxy, _maxx;
          short               _begy, _begx;
          short               _flags;
          short               _ch_off;
          bool                _clear;
          bool                _leave;
          bool                _scroll;
          char                **_y;
          short               *_firstch;
          short               *_lastch;
          struct _win_st      *_nextp, *_orig;
};

# define  _ENDLINE  001
# define  _FULLWIN  002
# define  _SCROLLWIN004
# define  _FLUSH              010
# define  _FULLLINE 020
# define  _IDLINE             040
# define  _STANDOUT 0200
# define  _NOCHANGE -1
```

_cury()[34] and _curx() are the current (y, x) coordinates for the window. New characters added to the screen are added at this point. _maxy() and _maxx() are the maximum values allowed for (_cury, _curx). _begy() and _begx() are the starting (y, x) coordinates on the terminal for the window, that is, the window's home. _cury(), _curx(), _maxy(), and _maxx() are measured relative to (_begy, _begx), not the terminal's home.

_clear() tells if a clear-screen sequence is to be generated on the next refresh() call. This is only meaningful for screens. The initial clear-screen for the first refresh() call is generated by initially setting clear to be TRUE for curscr, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. _leave() is TRUE if the current (y, x) coordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. _scroll() is TRUE if scrolling is allowed.

_y() is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the ith line, and

```
_y[i][j]
```

is the jth character on the ith line. _flags() can have one or more values or'd into it.

For windows that are not subwindows, _orig is NULL. For subwindows, it points to the main window to which the window is subsidiary. _nextp is a pointer in a circularly linked list of all the windows which are subwindows of the same main window, plus the main window itself.

_firstch and _lastch are malloc()ed arrays which contain the index of the first and last changed characters on the line. _ch_off is the x offset for the window in the _firstch and _lastch arrays for this window. For main windows, this is always 0; for subwindows it is the difference between the starting point of the main window and that of the subwindow, so that change markers can be set relative to the main window. This makes these markers global in scope.

All subwindows share the appropriate portions of _y(), _firstch, _lastch, and _insdel with their main window.

_ENDLINE says that the end of the line for this window is also the end of a screen. _FULLWIN says that this window is a screen. _SCROLLWIN indicates that the last character of this screen is at the lower right-hand corner of the terminal; that is, if a character was put there, the terminal would scroll. _FULLLINE says that the width of a line is the same as the width of the terminal. If _FLUSH

[34] All variables not normally accessed directly by the user are named with an initial '_' to avoid conflicts with the user's variables.

is set, it says that `fflush(stdout)` should be called at the end of each `refresh()`. `_STANDOUT` says that all characters added to the screen are in standout mode. `_INSDEL` is reserved for future use, and is set by `idlok()`. `_firstch` is set to `_NOCHANGE` for lines on which there has been no change since the last `refresh()`.

11.7. Example

Here is a simple example of how to use the package.

This example (`twinkle`) is intended to demonstrate the basic structure of a program using the screen updating sections of the package.

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion.

```
# include       <curses.h>
# include       <signal.h>

/*
 * the idea for this program was a product
 * of the imagination of Kurt Schoens.  Not
 * responsible for minds lost or stolen.
 */

# define        NCOLS   80
# define        NLINES  24
# define        MAXPATTERNS     4

struct locs {
        char    y, x;
};

typedef struct locs     LOCS;

LOCS    Layout[NCOLS * NLINES]; /* current board layout */

int     Pattern,                /* current pattern number */
        Numstars;               /* number of stars in pattern */

main() {

        char            *getenv();
        int             die();

        srand(getpid());                /* initialize random sequence */

        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);
```

```
        for (;;) {
                makeboard();            /* make the board setup */
                puton('*');             /* put on '*'s */
                puton(' ');             /* cover up with ' 's */
        }
}

/*
 * On program exit, move the cursor to the lower
 * left corner by direct addressing, since current
 * location is not guaranteed.  We lie and say we
 * used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}


/*
 * Make the current board setup.  It picks a random
 * pattern and calls ison() to determine if the
 * character is on that pattern or not.
 */
makeboard() {

        reg int         y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
                for (x = 0; x < NCOLS; x++)
                        if (ison(y, x)) {
                                lp->y = y;
                                lp++->x = x;
                        }
        Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {

        switch (Pattern) {
          case 0:       /* alternating lines */
                return !(y & 01);
```

```
            case 1:         /* box */
                  if (x >= LINES && y >= NCOLS)
                        return FALSE;
                  if (y < 3 || y >= NLINES - 3)
                        return TRUE;
                  return (x < 3 || x >= NCOLS - 3);
            case 2:         /* holy pattern! */
                  return ((x + y) & 01);
            case 3:         /* bar across center */
                  return (y >= 9 && y <= 15);
      }
      /* NOTREACHED */
}

puton(ch)
reg char        ch; {

      reg LOCS        *lp;
      reg int         r;
      reg LOCS        *end;
      LOCS            temp;

      end = &Layout[Numstars];
      for (lp = Layout; lp < end; lp++) {
            r = rand() % Numstars;
            temp = *lp;
            *lp = Layout[r];
            Layout[r] = temp;
      }

      for (lp = Layout; lp < end; lp++) {
            mvaddch(lp->y, lp->x, ch);
            refresh();
      }
}
```

12

System V `curses` and `terminfo`:

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a menu, divide a terminal screen into windows, or draw a display on the screen to help users enter and retrieve information from a database.

This tutorial explains how to use the System V `curses` and `terminfo` libraries to write screen management programs on a SunOS system. This package includes a library of C routines, a database of terminals and terminal capabilities, and a set of SunOS system support tools. To start you writing screen management programs as soon as possible, the tutorial does not attempt to cover every part of the package. For instance, it covers only the most frequently used routines and then points you to `curses`(3V) and `terminfo`(5V) in the *SunOS Reference Manual* for more information.

Because the routines are compiled C functions, you should be familiar with the C programming language before using `curses/terminfo`. You should also be familiar with the C language Standard I/O library.

This chapter has five sections: The *Overview* describes `curses`, `terminfo`, and the other components of the System V terminal information utilities package.

Working with `curses` Routines describes the basic routines making up the `curses`(3V) library. It covers the routines for writing to a screen, reading from a screen, and building *windows*.[35] It also covers routines for more advanced screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time. Many examples are included to show the effect of using these routines.

Working with `terminfo` Routines describes the routines in the `curses` library that deal directly with the `terminfo` database to handle certain terminal capabilities, such as programming function keys.

Working with the `terminfo` Database describes the `terminfo` database, related support tools, and their relationship to the `curses` library.

`curses` Program Examples includes six programs that illustrate various `curses` routines.

[35] Here the term *windows* refers to a region within a single terminal screen.

12.1. Overview

What is `curses`?

`curses`(3V) is the library of routines that you use to write screen management programs on the SunOS system. The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine `printw()` that behaves like `printf`(3V), and another named `getch()` that behaves like `getc`(3V). The automatic teller program at your bank might use `printw()` to print its menus and `getch()` to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the SunOS screen editor `vi`(1) might also use these and other `curses` routines.

The `curses` library is located in the file `/usr/5lib/libcurses.a`. To compile a program using routines in this library, you must use the System V optional `/usr/5bin/cc`(1V) command, and include the `-lcurses` on the command line so that the link editor can locate and load them:

```
/usr/5bin/cc file.c -lcurses -o file
```

The name `curses` comes from the cursor optimization that this library of routines provides. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you had designed a screen editor program with `curses` routines and edited the sentence

```
curses/terminfo is a great package for creating screens.
```

to read

```
curses/terminfo is the best package for creating screens.
```

the program would output only the string 'thebest in place of '.agreat The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal on which a `curses` program is run. This means that the `curses` library can do what is required to update any of a large number of different terminal types. It searches the `terminfo` database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your system. Fourth, it handles a large variety of terminals on which your program might be run.

Here's a simple `curses` program. It uses some of the basic `curses` routines to move a cursor to the middle of a screen and print the character string `BullsEye`. Each of these routines is described in the section *Working with `curses` Routines* later in this chapter. For now, just look at their names below and you will get an idea of what each of them does.

Figure 12-1 *A Simple* curses Program

```
#include <curses.h>

main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

What is terminfo?

terminfo refers to both of the following:

Terminfo Routines

This is a group of routines within the curses library for handling certain terminal capabilities. You can use these routines to program function keys (if your terminal has programmable keys), or write filters, for example. Shell programmers, as well as C programmers, can use the terminfo routines in their programs.

Terminfo Database

This is a database containing the descriptions of many terminals that can be used with curses programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that terminfo(5V) describes to create these files and the command tic(8V) to compile them.

The compiled files are normally located in the directories /usr/share/lib/terminfo/?. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for a virtual terminal emulator is normally located in the file /usr/share/lib/terminfo/v/virtual.

Here is a simple shell script that uses the `terminfo` database.

Figure 12-2 *A Shell Script Using* `terminfo` *Routines*

```
#   Clear the screen and show the 0,0 position.
#
tput clear
tput cup 0 0            # or tput home
echo "<- this is 0 0"
#
#   Show the 5,10 position.
#
tput cup 5 10
echo "<- this is 5 10"
```

How `curses` and `terminfo` Work Together

A screen management program with `curses` routines refers to the `terminfo` database at run time to obtain the information it needs about the terminal being used.

For example, suppose you are using a virtual terminal emulator to display the simple "BullsEye" program shown above. To execute properly, the program needs to know how many lines and columns the terminal screen has, in order to print the `BullsEye` in the middle of it. The description of the `ansi` terminal type in the `terminfo` database contains these values. All the `curses` program needs to know beforehand is the name of the terminal type. This is generally set automatically when you log in.

Other Components of the Terminal Information Utilities Package

Here is a complete list of the components discussed in this tutorial:

`captoinfo`(8V)
: a tool for converting terminal descriptions developed on earlier releases of the SunOS system to `terminfo` descriptions

`curses`(3V)
: the `curses` library

`infocmp`(8V)
: a tool for printing and comparing compiled terminal descriptions

`tabs`(1V)
: a tool for setting non-standard tab stops

`terminfo`(5V)
: the System V terminal information database

`tic`(8V)
: a tool for compiling terminal descriptions for the `terminfo` database

`tput`(1V)
: a tool for initializing the tab stops on a terminal and for outputting the value of a terminal capability

12.2. Working with curses Routines

This section describes the basic curses routines for creating interactive screen management programs. It begins by describing the routines and other program components that every curses program needs to work properly. Then it tells you how to compile and run a curses program. Finally, it describes the most frequently used curses routines that

- write output to and read input from a terminal screen
- control the data output and input — for example, to print output in bold type or prevent it from echoing (printing back on a screen)
- manipulate multiple screen images (windows)
- draw simple graphics
- manipulate soft labels on a terminal screen
- send output to and accept input from more than one terminal.

To illustrate the effect of using these routines, we include simple example programs as the routines are introduced. We also refer to a group of larger examples located in the section curses *Program Examples* in this chapter. These larger examples are more challenging; some make use of routines not discussed here.

What Every curses Program Needs

All curses programs need to include the header file <curses.h> and call the routines initscr(), refresh() or similar related routines, and endwin().

The Header File <curses.h>

The header file <curses.h> defines several global variables and data structures and defines several curses routines as macros.

To begin, let's consider the variables and data structures defined. <curses.h> defines all the parameters used by curses routines. It also defines the integer variables LINES and COLS; when a curses program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine initscr() described below. The header file defines the constants OK and ERR, too. Most curses routines have return values; the OK value is returned if a routine is properly completed, and the ERR value if some error occurs.

LINES and COLS are external (global) variables that represent the size of a terminal screen. The environment variables, LINES and COLUMNS, may be set in a user's shell environment; a curses program uses the environment variables to determine the size of a screen.

For more information about these variables, see *The Routines* initscr(), refresh(), *and* endwin() and *More about* initscr() *and Lines and Columns*, below.

Now let's consider the macro definitions. The <curses.h> header file defines many curses routines as macros that call (other macros or) curses routines. The line

```
#define refresh() wrefresh(stdscr)
```

shows when refresh is called, it is expanded to call the curses routine

`wrefresh()`. The latter routine, in turn, calls the two `curses` routines `wnoutrefresh()` and `doupdate()`. Many other macros also combine two or three routines together to achieve a particular result.

Macro expansion in `curses` programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about `<curses.h>`: it automatically includes `<stdio.h>` and the `<termio.h>`, terminal driver interface file. Including either file again in a program is redundant, but harmless.

The Routines `initscr()`, `refresh()`, and `endwin()`

The routines `initscr()`, `refresh()`, and `endwin()` initialize a terminal screen to an "in `curses` state," update the contents of the screen, and restore the terminal to an "out of `curses` state," respectively. Use the simple program that we introduced earlier to learn about each of these routines:

Figure 12-3 `initscr()`, `refresh()`, *and* `endwin()` *in a Program*

```
#include <curses.h>

main()
{
   initscr();      /* initialize terminal settings and <curses.h>
                      data structures and variables */

   move( LINES/2 - 1, COLS/2 - 4 );
   addstr("Bulls");
   refresh();      /* send output to (update) terminal screen */
   addstr("Eye");
   refresh();      /* send more output to terminal screen */
   endwin();       /* restore all terminal settings */
}
```

A `curses` program usually starts by calling `initscr()`; the program should call `initscr()` only once. Using the environment variable `TERM` as the section *How* `curses` *and* `terminfo` *Work Together* describes, this routine determines what terminal is being used. It then initializes all the declared data structures and other variables from `<curses.h>`. For example, `initscr()` would initialize `LINES` and `COLS` for the sample program on whatever terminal it was run. If a virtual terminal emulator were to be used, this routine would initialize `LINES` to 24 and `COLS` to 80. Finally, this routine writes error messages to `stderr` and exits if errors occur.

During the execution of the program, output and input is handled by routines like `move()` and `addstr()` in the sample program. For example,

```
move( LINES/2 - 1, COLS/2 - 4 );
```

says to move the cursor to the left of the middle of the screen. Then the line

```
addstr("Bulls");
```

says to write the character string `Bulls`. With a virtual terminal, these routines would position the cursor and write the character string at (11,36).

All curses routines that move the cursor move it from its home position in the upper left corner of a screen. The (LINES,COLS) coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the more common 'x,y' order of screen (or graph) coordinates.

The –1 in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like move() and addstr() do not actually change a physical terminal screen when they are called. The screen is updated only when refresh() is called. Before this, an internal representation of the screen called a *window* is updated. This is a very important concept, which we discuss below under *More about* refresh() *and Windows*.

Finally, a curses program ends by calling endwin(). This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling a curses Program

You compile programs that include curses routines as C language programs using the /usr/5bin/cc command, which invokes the C compiler.

The routines are stored in the library /usr/5lib/libcurses.a. To direct the link editor to search this library, you must use the –l option with the cc command.

The general command line for compiling a curses program follows:

```
/usr/5bin/cc file.c -lcurses -o file
```

file.c is the name of the source program; and *file* is the resulting executable program.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, initscr() sets the variables LINES and COLS. These variables are set from the terminfo variables lines and columns. These, in turn, are set from the values in the terminfo database, unless overridden by the window size obtained by the TIOCGWINSZ ioctl(2) request. If that size is zero, the values of the environment variables LINES and COLUMNS are used.

More about refresh() and Windows

As mentioned above, curses routines do not update a terminal until refresh() is called. Instead, they write to an internal representation of the screen called a *window*. When refresh() is called, the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like the buffer used by vi(1). When you invoke vi to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the w or ZZ command. Similarly, when you invoke a screen program made up of curses routines, they change the contents of a window. The changes become part of the current terminal screen only when refresh() is called.

<curses.h> supplies a default window named stdscr (standard screen), which is the size of the current terminal's screen, for all programs using curses routines. The header file defines stdscr to be of the type WINDOW*, a pointer to a C structure which you can think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in stdscr. When refresh() is called, it compares the two screen images and sends a stream of characters to the terminal that make the current screen look like stdscr. A curses program considers

many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is on the window. It optimizes output by printing as few characters as is possible. The following figure illustrates what happens when you execute the "BullsEye" curses program.

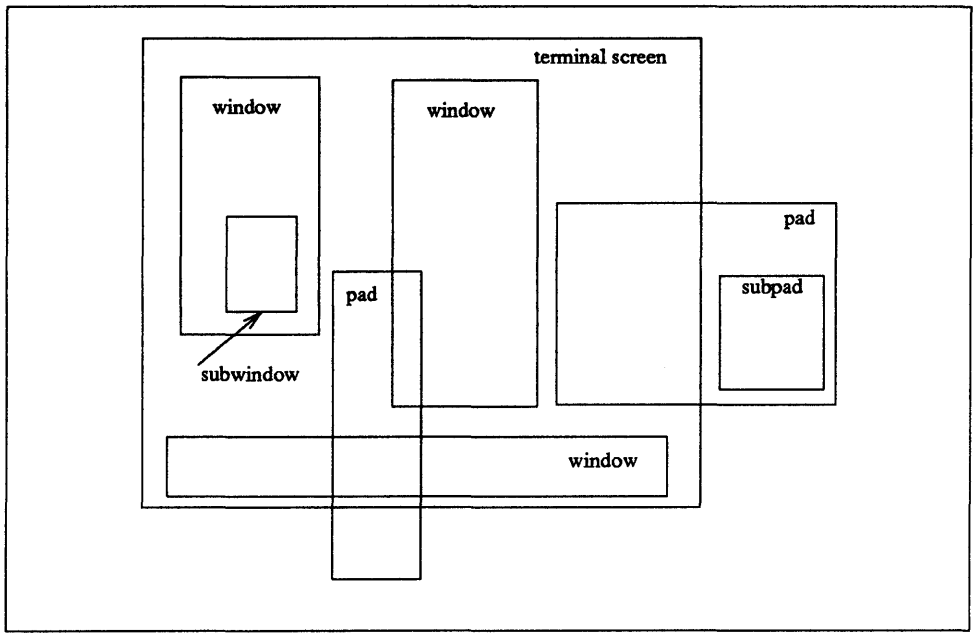
You can create other windows and use them instead of `stdscr`. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window.

It is possible to subdivide a screen into many windows, refreshing each one of them as desired. When windows overlap, the contents of the current screen show the most recently refreshed window. It is also possible to create a window within a window; the smaller window is called a subwindow. Assume that you are designing an application that uses forms, for example, an expense voucher, as a user interface. You could use subwindows to control access to certain fields on the form.

Some `curses` routines are designed to work with a special type of window called a *pad*. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

The illustration below represents what a pad, a subwindow, and some other windows might look like in comparison to a terminal screen.

Figure 12-4 *Multiple Windows and Pads Mapped to a Terminal Screen*



The section *Building Windows and Pads*, later in this chapter, describes the routines you use to create and use them.

Simple Output and Input

Output

The routines that curses provides for writing to stdscr are similar to those provided by the stdio(3V) library for writing to a file. They let you:

- write a character at a time — addch()
- write a string — addstr()
- format a string from a variety of input arguments — printw()
- move a cursor or move a cursor and print character(s) — move(), mvaddch(), mvaddstr(), mvprintw()
- clear a screen or a part of it — clear(), erase(), clrtoeol(), clrtobot()

Following are descriptions and examples of these routines.

The curses library provides its own set of output and input functions. You should not use other I/O routines or system calls, like read(2) and write(2), in a curses program. They may cause undesirable results when you run the program.

addch() — Write a single character to stdscr

```
#include <curses.h>
int addch(ch)
chtype ch;
```

addch() is a macro that writes a single character to stdscr. The character is of the type chtype, which is defined in <curses.h>. chtype contains both data and attributes (see *Output Attributes* in this chapter for information about attributes); when working with variables of this type, make sure you declare them as chtype, and not as the underlying data type (for example, short) of chtype. This will ensure future compatibility.

addch() does some character translations. For example, it maps the NEWLINE character to a clear-to-end-of-line, and moves the cursor to the next line. It maps the TAB character to an appropriate number of blanks. It maps other control characters to the appropriate '^*X*' notation.

addch() normally returns OK. The only time addch() returns ERR is after adding a character to the lower right-hand corner of a window that does not scroll.

Example:

```
#include <curses.h>

main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

produces:

```
a


$
```

Also see the `show` program under `curses` *Example Programs* later in this chapter.

`addstr()` — write a string of characters to `stdscr`

```
#include <curses.h>

int addstr(str)
char *str;
```

`addstr()` is a macro that follows the same translation rules as `addch()`; it calls `addch()` to write each character. `addstr()` returns `OK` on success and `ERR` on error.

For an example, refer to the "BullsEye" program, above.

`printw()` — formatted printing on `stdscr`

```
#include <curses.h>

int printw(fmt [,arg...])
char *fmt
```

Like `printf`, `printw()` takes a format string and a variable number of arguments. Like `addstr()`, `printw()` calls `addch()` to write the string. `printw()` returns `OK` on success and `ERR` on error.

Example:

```
#include <curses.h>

main()
{
   char* title = "Not specified";
   int no = 0;

   initscr();
   printw("%s is not in stock.\n", title);
   printw("Please ask the cashier to order %d for you.\n", no);
   refresh();
   endwin();
}
```

produces:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.


$
```

`move()` — position the cursor for `stdscr`

```
#include <curses.h>

int move(y, x);
int y, x;
```

`move()` positions the cursor for `stdscr` at the given row `y` and the given column `x`.

Notice that `move()` takes the `y` coordinate *before* the `x` coordinate. The upper left-hand coordinates for `stdscr` are (0,0), the lower right-hand (`LINES` – 1, `COLS` – 1). See the section `initscr()`, `refresh()`, *and* `endwin()` for more information.

`move()` returns `OK` on success and `ERR` on error. Trying to move to a screen position of less than (0,0) or more than (`LINES` - 1, `COLS` - 1) causes an error.

Example:

```
#include <curses.h>

main()
{
   initscr();
   addstr("Cursor should be here --> if move() works.");
   printw("\n\n\nPress <CR> to end test.");
   move(0,25);
   refresh();
   getch();     /* Gets <CR>; discussed below. */
   endwin();
}
```

produces:

```
Cursor should be here -->■if move() works.


Press <CR> to end test.
```

After you press RETURN, the screen looks like:

```
Cursor should be here -->


Press <CR> to end test.
$■
```

See the `scatter` program under `curses` *Program Examples* in this chapter for another example.

`mvaddch` — move and print a character

```
#include <curses.h>
int mvaddch( y, x, ch )
```

`mvaddch()` is a macro that moves the cursor to a given position and prints a character.

`mvaddstr` — move and print a string

```
#include <curses.h>
int mvaddstr( y, x, str )
```

`mvaddstr()` is a macro that moves the cursor to a given position and prints a string of characters.

mvprintw — move and print a formatted string

```
#include <curses.h>
int mvprintw( y, x, fmt [,arg]... )
```

mvprintw() is a macro that moves the cursor to a given position and prints a formatted string. of using move().

clear() and erase() — clear the screen

```
#include <curses.h>
int clear()
int erase()
```

clear() and erase() are macros that convert stdscr to all blanks. clear() assumes that the screen may have garbage that it doesn't know about; it first calls erase() and then clearok(), which clears the physical screen completely on the next call to refresh(). initscr() automatically calls clear().

clear() always returns OK; erase() returns no useful value.

clrtoeol() and clrtobot() — partial screen clears

```
#include <curses.h>
int clrtoeol()
int clrtobot()
```

clrtoeol() and clrtobot() are macros that clear a portion of the screen. clrtoeol() changes the remainder of a line to all blanks. clrtobot() changes the remainder of a screen to all blanks. Both start with the current cursor position inclusive.

Neither returns any useful value.

Example:

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtobot();
    refresh();
    endwin();
}
```

produces:

```
Press <CR> to delete from here■to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh()`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press RETURN:

```
Press <CR> to delete from here

$■
```

See the `show` and `two` programs under `curses` *Example Programs* for examples of `clrtoeol()`.

Input

`curses` routines for reading from the current terminal are similar to those provided by the `stdio`(3V) library for reading from a file. They let you

- read a character at a time — `getch()`
- read a NEWLINE-terminated string — `getstr()`
- parse input, converting and assigning selected data to an argument list — `scanw()`

The primary routine is `getch()`, which processes a single input character and then returns that character. This routine is like the C library routine `getchar()`(3V) except that it makes several terminal- or system-dependent options available that are not possible with `getchar()`. For example, you can use `getch()` with the `curses` routine `keypad()`, which allows a `curses` program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key.

`getch()` — read a single character from the current terminal

```
#include <curses.h>
int getch()
```

`getch()` is a macro that returns the value of the character or `ERR` on 'end of file', receipt of signals, or non-blocking read with no input.

See the discussions about `echo()`, `noecho()`, `cbreak()`, `nocbreak()`, `raw()`, `noraw()`, `halfdelay()`, `nodelay()`, and `keypad()` below.

Example:

```
#include <curses.h>

main()
{
   int ch;

   initscr();
   cbreak();          /* Explained later in the section "Input Options" */
   addstr("Press any character:  ");
   refresh();
   ch = getch();
   printw("\n\n\nThe character entered was a '%c'.\n", ch);
   refresh();
   endwin();
}
```

The first `refresh()` sends the `addstr()` character string from `stdscr` to the terminal:

```
Press any character:
```

Then assume that a `w` is typed at the keyboard. `getch()` accepts the character and assigns it to `ch`. Finally, the second `refresh()` is called:

```
Press any character:  w


The character entered was a 'w'.


$
```

For another example of `getch()`, see the `show` program under `curses` *Example Programs*.

`getstr()` — read character string into a buffer

```
#include <curses.h>

int getstr(str)
char *str;
```

`getstr()` is a macro that calls `getch()` to read a string of characters into a buffer, until a RETURN, NEWLINE, or ENTER key is received from `stdscr`. `getstr()` does not check for buffer overflow.

`getstr()` returns `ERR` if `getch()` returns `ERR`; otherwise it returns `OK`.

See the discussions about `echo()`, `noecho()`, `cbreak()`, `nocbreak()`, `raw()`, `noraw()`, `halfdelay()`, `nodelay()`, and `keypad()` below.

Example:

```
#include <curses.h>

main()
{
char str[256];

   initscr();
   cbreak();       /* Explained later in the section "Input Options" */
   addstr("Enter a character string terminated by <CR>:\n\n");
   refresh()
   getstr(str);
   printw("\n\n\nThe string entered was \n'%s'\n", str);
   refresh();
   endwin();
}
```

If you enter the string 'I enjoy learning about the SunOS system', the final screen (after entering RETURN) would appear as:

```
Enter a character string terminated by <CR>:

I enjoy learning about the SunOS system


The string entered was
'I enjoy learning about the SunOS system.'

$
```

`scanw()` — formatted input conversion

```
#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;
```

Like `scanf`(3V), `scanw()` uses a format string to convert input words and assign them to a variable number of arguments. `scanw()` returns the same values as `scanf()`.

See `scanf`(3V) for more information.

Example:

```
#include <curses.h>

main()
{
   char string[100];
   float number;

   initscr();
   cbreak();           /* Explained later in the  */
   echo();             /* section "Input Options" */
   addstr("Enter a number and a string separated by a comma: ");
   refresh();
   scanw("%f,%s",&number,string);
   clear();
   printw("The string was \"%s\" and the number was %f.",string,number);
   refresh();
   endwin();
}
```

Notice the two calls to `refresh()`. The first call updates the screen with the character string passed to `addstr()`, the second with the string returned from `scanw()`. Also notice the call to `clear()`. Assume you entered the following when prompted: `2,twin`. After running this program, your terminal screen would appear, as follows:

```
The string was "twin" and the number was 2.000000.

$
```

Controlling Output and Input

Output Attributes

When we talked about `addch()`, we said that it writes a single character of the type `chtype` to `stdscr`. `chtype` has two parts: a part with information about the character itself, and another part with information about a set of attributes associated with the character. These attributes allow a character to be printed in reverse video, bold, underlined, and so on.

`stdscr` always has a set of current attributes that it associates with each character as it is written. However, using the routine `attrset()` and the related `curses` routines described below, you can change the current attributes. Below is a list of the attributes and what they mean.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, a `curses` program attempts to find a substitute attribute. If none is possible, the attribute is ignored.

`A_BLINK`	blinking
`A_BOLD`	extra bright or bold
`A_DIM`	half bright
`A_REVERSE`	reverse video
`A_STANDOUT`	a terminal's best highlighting mode
`A_UNDERLINE`	underlining
`A_ALTCHARSET`	alternate character set

(See the section *Drawing Lines and Other Graphics*, below, for more information about these attributes.)

To use these attributes, you must pass them as arguments to `attrset()` and related routines; they can also be `OR`'ed with the bitwise `OR (|)` to `addch()`.

Let's consider a use of one of these attributes. To display a word in bold, use the following code:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
refresh();
```

Attributes can be turned on singly, such as `attrset(A_BOLD)` in the example, or in combination. To turn on blinking bold text, for example, you would use `attrset(A_BLINK | A_BOLD)`. Individual attributes can be turned on and off with the `curses` routines `attron()` and `attroff()` without affecting other attributes. `attrset(0)` turns all attributes off.

Notice the attribute called `A_STANDOUT`. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the `A_STANDOUT` attribute is recommended. Two convenient functions, `standout()` and `standend()` can be used to turn on and off this attribute. `standend()`, in fact, turns off all attributes.

Bit Masks

In addition to the attributes listed above, there are two bit masks called `A_CHARTEXT` and `A_ATTRIBUTES`. You can use these bit masks with the `curses` function `inch()` and the C logical `AND (&)` operator to extract the character or attributes of a position on a terminal screen. See the discussion of `inch()` for more information.

Following are descriptions of `attrset()` and the other `curses` routines that you can use to manipulate attributes.

attron(), attrset(), and attroff() — set or modify attributes

```
#include <curses.h>

int attron( attrs )
chtype attrs;

int attrset( attrs )
chtype attrs;

int attroff( attrs )
chtype attrs;
```

attron() turns on the requested attribute attrs in addition to any that are currently on. *Attrs* is of the type chtype and is defined in <curses.h>.

attrset() turns on the requested attributes attrs instead of any that are currently turned on.

attroff() turns off the requested attributes, *attrs*, if they are on.

Attributes may be combined using the bitwise OR (|).

All return OK.

Example:
See the highlight program under curses *Example Programs*, below.

standout() and standend() — highlight with preferred attribute

```
#include <curses.h>

int standout()
int standend()
```

standout() turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to attron(A_STANDOUT).

standend() turns off all attributes. This routine is equivalent to attrset(0).

Both always return OK.

Example:
See the highlight program under curses *Example Programs*, below.

Bells, Whistles, and Flashing Lights

Occasionally, you may want to get a user's attention. Two curses routines were designed to help you do this. They let you ring the terminal's bell and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine beep() can be called when an audible bell is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to beep() will flash the screen.)

`beep()` and `flash()` — ring bell or flash screen

```
#include <curses.h>

int flash()
int beep()
```

`flash()` tries to flash the terminal screen, if possible, otherwise it tries to ring the terminal bell.

`beep()` tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.

Neither returns any useful value.

Input Options

The SunOS system does a considerable amount of processing on input before an application ever sees a character; amongst other things, it:

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character, typically DELETE and a line kill character, typically CTRL-U (control-U)
- interprets a CTRL-D as end-of-file (EOF) character.
- interprets interrupt and quit characters
- strips the character's parity bit
- translates RETURN characters to NEWLINEs.

Because a `curses` program maintains total control over the screen, `curses` turns off echoing; it does the echoing itself. For an interactive screen, you may not want the system to process characters in the standard way. Some `curses` routines, `noecho()` and `cbreak()`, for example, have been designed so that you can alter the standard character processing. Using these routines in an application controls how input is interpreted.

Every `curses` program accepting input should set some input options so that when the program starts running, the terminal on which it runs will be in `cbreak()`, `raw()`, `nocbreak()`, or `noraw()` mode. Although the `curses` program starts up in `echo()` mode, as shown below, none of the other modes are guaranteed.

The combination of `noecho()` and `cbreak()` is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The `curses` routine `noecho()` is designed for this purpose. However, when `noecho()` turns off echoing, normal erase and kill processing is still on. Using the routine `cbreak()` causes these characters to be uninterpreted.

Figure 12-5 *Input Option Settings for* curses Programs

Input Options	Characters Interpreted	Uninterpreted
Normal 'out of curses state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal curses 'start up state'	echoing (simulated)	All else undefined.
cbreak() and echo()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and noecho()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and noecho()	break, quit stripping erase, kill EOF	echoing
nocbreak() and echo()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of cbreak())		break, quit stripping

Do not use the combination nocbreak() and noecho(). If you use it in a program and also use getch(), the program will go in and out of cbreak() mode to get each character. Depending on the state of the terminal driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted above, you can use the curses routines noraw(), halfdelay(), and nodelay() to control input. These routines are described in curses(3V).

`echo()` and `noecho()` — turn echoing on and off

```
#include <curses.h>

int echo()
int noecho()
```

`echo()` turns on echoing of characters by `curses` as they are read in. This is the initial setting.

`noecho()` turns off the echoing.

Neither returns any useful value.

`curses` programs may not run properly if you turn on echoing with `noc-break()`. After you turn echoing off, you can still echo characters with `addch()`.

Examples:
See the `editor` and `show` programs under `curses` *Program Examples*, below.

`cbreak()` and `nocbreak()` — turn "break for each character" on or off

```
#include < curses.h >
int cbreak()
int nocbreak()
```

`cbreak()` turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.

`nocbreak()` returns to normal 'line at a time' processing. This is typically the initial setting.

Neither returns any useful value.

A `curses` program may not run properly if `cbreak()` is turned on and off within the same program or if the combination `nocbreak()` and `echo()` is used.

Example:
See the `editor` and `show` programs under `curses` *Program Examples*.

Building Windows and Pads

The section above entitled *More about* `refresh()` *and Windows* explained what windows and pads are and why you might want to use them. This section describes the `curses` routines you use to manipulate and create windows and pads.

Window Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with `stdscr`. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter `w` at the beginning of the name of a `stdscr` routine and adding the window name as the first parameter. For example, `addch('c')` would become `waddch(mywin, 'c')` if you wanted to write the character `c` to the window `mywin`. Here's a list of the window (or `w`) versions of the output routines discussed in *Getting Simple Output and Input.*

```
waddch(win, ch)
mvwaddch(win, y, x, ch)
waddstr(win, str)
mvwaddstr(win, y, x, str)
wprintw(win, fmt [, arg ...])
mvwprintw(win, y, x, fmt [, arg ...])
wmove(win, y, x)
wclear(win) and werase(win)
wclrtoeol(win) and wclrtobot(win)
wrefresh()
```

You can see from their declarations that these routines differ from the versions that manipulate `stdscr` only in their names and the addition of a *win* argument. Notice that the routines whose names begin with `mvw` take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See `curses`(3V) for more information about these routines, or the versions of the input routines `getch`, `getstr()`, and so on that you should use with windows.

All `w` routines can be used with pads except for `wrefresh()` and `wnoutrefresh()`. In place of these two routines, you have to use `prefresh()` and `pnoutrefresh()` with pads.

The Routines `wnoutrefresh()` and `doupdate()`

If you recall from the earlier discussion about `refresh()`, we said that it sends the output from `stdscr` to the terminal screen. We also said that it was a macro that expands to `wrefresh(stdscr)` (see *What Every* `curses` *Program Needs* and *More about* `refresh()` *and Windows*).

The `wrefresh()` routine is used to send the contents of a window (`stdscr` or one that you create) to a screen; it calls the routines `wnoutrefresh()` and `doupdate()`. Similarly, `prefresh()` sends the contents of a pad to a screen by calling `pnoutrefresh()` and `doupdate()`.

Using `wnoutrefresh()`—or `pnoutrefresh()` (this discussion will be limited to the former routine for simplicity)—and `doupdate()`, you can update terminal screens with more efficiency than using `wrefresh()` by itself. `wrefresh()` works by first calling `wnoutrefresh()`, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling `wnoutrefresh()`, `wrefresh()` then calls `doupdate()`, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling `wrefresh()` will result in alternating calls to `wnoutrefresh()` and `doupdate()`, causing several bursts of output to a screen. However, by calling `wnoutrefresh()` for each window and then `doupdate()` only once, you can minimize the total number of characters transmitted and the processor time used. The sample program below uses only one `doupdate()`.

```
#include <curses.h>

main()
{
    WINDOW *w1, *w2;

    initscr();
    w1 = newwin(2,6,0,3);
    w2 = newwin(1,4,5,4);
    waddstr(w1, "Bulls");
    wnoutrefresh(w1);
    waddstr(w2, "Eye");
    wnoutrefresh(w2);
    doupdate();
    endwin();
}
```

Notice from the sample that you declare a new window at the beginning of a `curses` program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named w1 and w2 with the routine `newwin()` according to certain specifications.

New Windows

Following are descriptions of the routines `newwin()` and `subwin()`, which you use to create new windows. For information about creating new pads with `newpad()` and `subpad()`, see `curses`(3V).

`newwin()` — open and return a pointer to new window

```
#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

`newwin()` returns a pointer to a new window with a new data area. The variables `nlines` and `ncols` give the size of the new window. `begin_y` and `begin_x` give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

Example:

See the `window` program under `curses` *Program Examples*.

subwin()

```
#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

subwin() returns a new window that points to a section of another window, orig. nlines and ncols give the size of the new subwindow. begin_y and begin_x give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.

Subwindows and original windows can accidentally overwrite one another.

Subwindows of subwindows are not allowed.

Example:

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');     /* See the curses(3V) manual page for box() */
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of ws around the stdscr (the sides of your terminal screen) and a border of s characters around the subwindow sub when it is run.

Using Advanced curses Features

Knowing how to use the basic curses routines to get output and input and to work with windows, you can design screen management programs that meet the needs of many users. The curses library, however, has routines that let you do more in a program than handle I/O and multiple windows. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single curses program.

You should be comfortable using the routines previously discussed in this chapter and the other routines for I/O and window manipulation discussed on the curses(3V) manual page before you try to use the advanced curses features.

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs, or graphic symbols). You can use this character set in `curses` programs. `curses` use the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in a `curses` program, pass a set of variables whose names begin with `ACS_` to the `curses` routine `waddch()` or a related routine. For example, `ACS_ULCORNER` is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, `ACS_ULCORNER`'s value is the terminal's character for that glyph, ORed (`|`) with the bit-mask `A_ALTCHARSET`. If no line-drawing character is available for that glyph, a standard *ASCII* character that approximates the glyph is stored in its place. For example, the default character for `ACS_HLINE`, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard `ACS_` names and their defaults are listed in `curses`(3V).

Part of an example program that uses line drawing characters follows. The example uses the `curses` routine `box()` to draw a box around a menu on a screen. `box()` uses the line drawing characters by default or when | (the pipe) and – are chosen. (See `curses`(3V).) Up and down more indicators are drawn on the box border (using `ACS_UARROW` and `ACS_DARROW`) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lowercase letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The curses library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for a curses program to make use of them.

Let's briefly discuss most of the curses routines needed to use soft labels: slk_init(), slk_set(), slk_refresh() and slk_noutrefresh(), slk_clear, and slk_restore.

When you use soft labels in a curses program, you have to call the routine slk_int() before initscr(). This sets an internal flag for initscr() to look at that says to use the soft labels. If initscr() discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of stdscr to use for the soft labels. The size of stdscr and the LINES variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the LINES and COLS variables, will continue to run as if the line had never existed on the screen.

slk_init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from stdscr. The choices are between a 3-2-3 arrangement, and a 4-4 arrangement. The curses routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine slk_set() takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left-justified, 1 = centered, and 2 = right-justified).

The routine slk_noutrefresh() is comparable to wnoutrefresh() in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a wrefresh() commonly follows, slk_noutrefresh() is the function that is most commonly used to output the labels.

Just as wrefresh() is equivalent to a wnoutrefresh() followed by a doupdate(), so too the function slk_refresh() is equivalent to a slk_noutrefresh() followed by a doupdate().

To prevent the soft labels from getting in the way of a shell escape, slk_clear() may be called before doing the endwin(). This clears the soft labels off the screen and does a doupdate(). The function slk_restore() may be used to restore them to the screen. See the curses(3V) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

A `curses` program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the `curses` library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the filename and terminal-type of each terminal. The standard method, checking `TERM` in the environment, does not work, because each process can only examine its *own* environment.

Another problem you might face is that of multiple programs reading from one `tty` line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on its line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

A `curses` program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary `curses` routines.

References to terminals in a `curses` program have the type `SCREEN*`. A new terminal is initialized by calling `newterm(`*`type, outfd, infd`*`)`. `newterm ()` returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a `stdio`(3V) file pointer (`FILE*`) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to `initscr()`, which calls `newterm(getenv(''TERM''), stdout, stdin)`.

To change the current terminal, call `set_term(`*`sp`*`)` where *sp* is the screen reference to be made current. `set_term()` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm()`. Options such as `cbreak()` and `noecho()` must be set separately for each terminal. The functions `endwin()` and `refresh()` must be called separately for each terminal. The figure below shows a typical scenario to output a message to several terminals.

Figure 12-6 *Sending a Message to Several Terminals*

```
for (i=0; i<nterm; i++)
{
   set_term(terms[i]);
   mvaddstr(0, 0, "Important message");
   refresh();
}
```

See the `two` program under `curses` *Program Examples* for a more complete example.

12.3. Working with `terminfo` Routines

Some programs need to use lower-level routines than those offered by the `curses` routines. For such programs, the `terminfo` routines are offered. They do not manage your terminal screen, but rather, give you access to strings and capabilities which you can use yourself to manipulate the terminal.

`terminfo` routines should not be used directly, except in the circumstances noted at right; the equivalent `curses` routines protect your program from the idiosyncracies of physical terminals. When you use the `terminfo` routines, you must deal with them yourself. Also, these low-level routines may change, rendering programs that rely on them obsolete.

There are three circumstances when it is proper to use `terminfo` routines directly. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the `terminfo` routines is worthwhile. The third is when you are writing a special-purpose tool that sends a special string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line.

Otherwise, you are discouraged from using these routines: the higher level `curses` routines make your program more portable to other SunOS systems, and to a wider class of terminals.

What Every `terminfo` Program Needs

A `terminfo` program typically includes the header files and routines shown below:

Figure 12-7 *Typical Framework of a* `terminfo` Program

```
#include <curses.h>
#include <term.h>
...
   setupterm( (char*)0, 1, (int*)0 );
   ...
   putp(clear_screen);
   ...
   reset_shell_mode();
   exit(0);
```

The header files <curses.h> and <term.h> are required because they contain the definitions of the strings, numbers, and flags used by the terminfo routines. setupterm() takes care of initialization. Passing this routine the values (char*)0, 1, and (int*)0 invokes reasonable defaults. If setupterm() can't figure out what kind of terminal you are on, it prints an error message and exits. reset_shell_mode() performs functions similar to endwin() and should be called before a terminfo program exits.

A global variable like clear_screen is defined by the call to setupterm(). It can be output using the terminfo routines putp() or tputs(), which gives a user more control. This string should not be directly output to the terminal using the C library routine printf(3V), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the xon/xoff flow control protocol.

At the terminfo level, the higher level routines like addch() and getch() are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see terminfo(5V); see curses(3V) for a list of all the terminfo routines.

Compiling and Running a terminfo Program

The general command line for compiling, and the guidelines for running a program with terminfo routines are the same as those for compiling any other curses program.

An Example terminfo Program

The example program, termhl, shows a simple use of terminfo routines. It is a version of the highlight program (see curses *Program Examples*) that does not use the higher level curses routines. termhl can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;      /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }
```

```
    if (argc == 2)
    {
        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
            perror(argv[1]);
            exit(2);
        }
    }
    else
    {
        fd = stdin;
    }
    setupterm((char*)0, 1, (int*)0);

    for (;;)
    {
        c = getc(fd);
        if (c == EOF)
        break;
        if (c == '\')
        {
            c2 = getc(fd);
            switch (c2)
            {
                case 'B':
                tputs(enter_bold_mode, 1, outch);
                continue;
                case 'U':
                tputs(enter_underline_mode, 1, outch);
                ulmode = 1;
                continue;
                case 'N':
                tputs(exit_attribute_mode, 1, outch);
                ulmode = 0;
                continue;
            }
            putch(c);
            putch(c2);
        }
        else
            putch(c);

    }
    fclose(fd);
    fflush(stdout);
    resetterm();
    exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
    int c;
{
    outch(c);
    if (ulmode && underline_char)
    {
```

```
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
    int c;
{
    putchar(c);
}
```

Let's discuss the use of the function `tputs` (*cap, affcnt, outc*) in this program to gain some insight into the `terminfo` routines. `tputs()` applies padding information. Some terminals have the capability to delay output. Their terminal descriptions in the `terminfo` database probably contain strings like `$<20>`, which means to pad for 20 milliseconds (see the following section *Specifying Capabilities*). `tputs` generates enough pad characters to delay for the appropriate time.

`tput()` has three parameters. The first parameter is the string capability to be output.

The second is the number of lines affected by the capability. Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.

The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls `putchar`. For these programs, the routine `putp` (*cap*) is a convenient abbreviation. `termhl` could be simplified by using `putp()`.

Now to understand why you should use the `curses` level routines instead of `terminfo` level routines whenever possible, note the special check for the `underline_char` capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. `termhl` keeps track of the current mode, and if the current character is supposed to be underlined, outputs `underline_char`, if necessary. Low level details such as this are precisely why the `curses` level is recommended over the terminfo level. `curses` takes care of terminals with different methods of underlining and other terminal functions. Programs at the `terminfo` level must handle such details themselves.

`termhl` was written to illustrate a typical use of the `terminfo` routines. It is more complex than it need be in order to illustrate some properties of `terminfo` programs. The routine `vidattr` (see `curses`(3V)) could have been

used instead of directly outputting enter_bold_mode, enter_underline_mode, and exit_attribute_mode. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

12.4. Working with the terminfo Database

The terminfo database describes the many terminals with which curses programs, as well as some SunOS system tools, like vi(1), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the terminfo database, related support tools, and their relationship to the curses library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already provided in the terminfo database. However, it is possible that you'll want to run a curses program on a terminal for which there is no existing description. In this case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.
2. Learn about, list, and define the known capabilities.
3. Compile the newly-created description entry.
4. Test the entry for correct operation.
5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier.

Naming the Terminal

The name of a terminal is the first information given in a terminfo terminal description. This string of names, assuming there is more than one name, is separated by vertical bars (|). The first name given should be the most common abbreviation for the terminal. The last name given is typically a verbose entry that fully identifies the terminal by make and model. The long name or "verbose" is typically the manufacturer's formal name for the terminal. Names between the first and last entries are known synonyms for the terminal name. All but the verbose name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description for a virtual terminal.

```
virtual|VIRTUAL|cbunix|cb-unix|cb-unix virtual terminal,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for a fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like `virtual` or `myterm`, for example. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a `-w`) version of our fictitious terminal would be described as `myterm-w`. `terminfo`(5V) describes mode indicators in greater detail.

Learning About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following wayss, type:

```
stty -echo; cat -vu
```

followed by the keys you want to test. To return to the shell and restore echo, type:

```
^D
stty echo
```

Note that `stty echo` is not displayed on the terminal screen.

Specifying Capabilities

Once you know the capabilities of your terminal, you have to provide them in your terminal description. Capability entries consist of a list of comma-separated fields containing the abbreviated `terminfo` name and, in some cases, the terminal's value for each capability. For example, `bel` is the abbreviated name for the beeping or ringing capability. On most terminals, a CTRL-G is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as `bel=^G,`.

The list of capabilities may continue across input lines as long as the continuation lines start with a white-space character, or consist of a comment. Comments can be included within the description by putting a # at the beginning of the line.

For a `curses` program to run on any given terminal, its description in the `terminfo` database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

The `terminfo`(5V) manual page has a complete list of the capabilities you can use in a terminal description.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters are given below.

This shows that a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as `cols#80,`.

= This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

^ This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as `^G`.

`\E \e` These characters followed by another character show an escape instruction. An entry of `\EC` would transmit to the terminal as [ESC-C.]

`\n` These characters provide a [NEWLINE] character sequence.

`\l` These characters provide a [LINEFEED] character sequence.

`\r` These characters provide a [RETURN] character sequence.

`\t` These characters provide a [TAB] character sequence.

`\b` These characters provide a [BACKSPACE] character sequence.

`\f` These characters provide a [FORMFEED] character sequence.

`\s` These characters provide a [SPACE] character sequence.

`\`*nnn* This is a character whose three-digit octal is *nnn* (*nnn* can be from one to three digits).

`$<`*n*`>` These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the brackets. The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay is to be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as `$<20*>`. See the `terminfo`(5V) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel=^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of `myterm`. We'll consider basic capabilities, screen-oriented capabilities, keyboard-entered capabilities, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated `terminfo` name for each capability in the parentheses following the capability description:

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (`am`).
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is `^G` (`bel`).
- An 80-column wide screen (`cols`).
- A 30-line long screen (`lines`).
- Use of xon/xoff protocol (`xon`).

By combining the name string with the capability descriptions that we now have, we get the following general `terminfo` database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
        am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal `myterm` has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A [RETURN] is a [CTRL-M] (`cr`).
- A cursor up one line motion is a [CTRL-K] (`cuu1`).
- A cursor down one line motion is a [CTRL-J] (`cud1`).
- Moving the cursor to the left one space is a [CTRL-H] (`cub1`).
- Moving the cursor to the right one space is a [CTRL-L] (`cuf1`).
- Entering reverse video mode is an [ESCAPE-D] (`smso`).
- Exiting reverse video mode is an [ESCAPE-Z] (`rmso`).
- A clear to the end of a line sequence is an [ESCAPE-K] and should have a 3-millisecond delay (`el`).

A terminal scrolls when receiving a [NEWLINE] at the bottom of a page (`ind`).

The revised terminal description for `myterm` including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (kbs).
- The up arrow key generates an ESCAPE-[A (kcuu1).
- The down arrow key generates an ESCAPE-[B (kcud1).
- The right arrow key generates an ESCAPE-[C (kcuf1).
- The left arrow key generates an ESCAPE-[D (kcub1).

The home key generates an ESCAPE-[H (khome).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters, such as those used to position a cursor on a screen, or to turn on a combination of video modes. To address a cursor, the cup capability is used and is passed two parameters: the row and column to address. String capabilities, such as cup and set attributes (sgr) capabilities, are passed arguments in a terminfo program by the tparm() routine.

The arguments to string capabilities are manipulated with special % sequences similar to those found in a call to printf(3V). In addition, many of the features found on a simple stack-based RPN calculator are available. cup, as noted above, takes two arguments: the row and column. sgr, takes nine arguments, one for each of the nine video attributes. See terminfo(5V) for the list and order of the attributes and further examples of sgr.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed with H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence ;r"ESCAPE-[6 would be output.

Integer arguments are pushed onto the stack with a %p sequence followed by the argument number, such as %p2 to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a %d sequence is used, exactly as in printf.

Our terminal's cup sequence is built up as follows:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
        am, bel=^G, cols#80, lines#30, xon,
        cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
        smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
        kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
        kcub1=\E[D, khome=\E[H,
        cup=\E[%i%p1%d;%p2%dH,
```

See terminfo(5V) for more information about parameter string capabilities.

Compiling the Description

The terminfo database entries are compiled using tic, the terminfo compiler command. This compiler translates terminfo source entries into the compiled format used by the terminfo and curses routines.

The source file for the source file is usually suffixed with .ti. For example, the description of myterm would be in a source file named myterm.ti. The compiled description of myterm would usually be placed in /usr/share/lib/terminfo/m/myterm, since the first letter in the description entry is m. Links would also be made to synonyms of myterm, for example, to /f/fancy. If the environment variable TERMINFO were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the TERMINFO directory. All programs using the entry would then look in the new directory for the description file if TERMINFO were set, before looking in the default /usr/share/lib/terminfo. The general format for the tic command is:

```
tic [-v] [-c] sourcefile
```

With the –v, verbose option, the compiler traces its actions and prints messages regarding its progress. The –c option checks for errors. tic(8V) compiles only one file at a time. The following command line shows how to compile the terminfo source file for myterm.

```
tic -v myterm.ti
```

Refer to tic(8V) for more information.

Testing the Description

Let's consider ways to test a terminal description. First, you can test it by setting the environment variable TERMINFO to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Or, you can use the tput(1V) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then tput sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the tput command is as follows:

tput [–T*type*] *capname*

The type of terminal you are requesting information about is identified with the –T*type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable TERM. The *capname* field is used to show what capability to output from the terminfo database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

```
tput clear
```

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols
```

tput(8V) contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the terminfo source directory. The infocmp(8V) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic oldvirtual.ti
TERMINFO=/tmp/new tic newvirtual.ti
infocmp -A /tmp/old -B /tmp/new -d virtual virtual
```

compares the old and new virtual entries.

To print out the `terminfo` source for the `virtual`, type:

```
infocmp -I virtual
```

Converting a `termcap` Description to a `terminfo` Description

The `terminfo` database is an alternative to the `termcap` database. Because of the many programs and processes that have been written with and for the `termcap` database, it is not feasible to do a complete conversion from `termcap` to `terminfo`. Since converting between the two requires experience with both, all entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The `captoinfo`(8V) command converts `termcap`(5) descriptions to `terminfo`(5V) descriptions. When a file is passed to `captoinfo`, it looks for `termcap` descriptions and writes the equivalent `terminfo` descriptions on the standard output. For example,

```
captoinfo /etc/termcap
```

converts the file `/etc/termcap` to `terminfo` source, preserving comments and other extraneous information within the file. The command line

```
captoinfo
```

looks up the current terminal in the `termcap` database, as specified by the `TERM` and `TERMCAP` environment variables and converts it to `terminfo`.

To convert a `terminfo` description into a `termcap` entry, use **`infocmp -C`**.

If you have been using cursor optimization programs with the `-ltermcap` or `-ltermlib` option in the `/usr/5bin/cc` command line, those programs should still be functional.

12.5. `curses` Program Examples

The following examples demonstrate uses of `curses` routines.

The `editor` Program

This program illustrates how to use `curses` routines to write a screen editor. For simplicity, `editor` keeps the buffer in `stdscr`; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the `move()`, `mvaddstr()`, `flash()`, `wnoutrefresh()` and `clrtoeol()` routines. These routines are all discussed in this chapter under *Working with `curses`* Routines.

Second, it also uses some `curses` routines that we have not discussed. For example, the function to write out a file uses the `mvinch()` routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the

number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the `insch()`, `delch()`, `insertln()`, and `deleteln()` routines. These functions insert and delete a character or line. See `curses`(3V) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

Since not all terminals have arrow keys, your `curses` programs will work with more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using `curses` routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking `editor` can type CTRL-L, causing the screen to be cleared and redrawn with a call to `wrefresh(curscr)`.

Finally, another important point is that the input command is terminated by CTRL-D, not the ESCAPE key. It is very tempting to use ESCAPE as a command, since it is one of the few special keys available on all keyboards. (RETURN and BREAK are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal, and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the ESCAPE key or whether a special key was pressed.

`editor` and other `curses` programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press ESCAPE, then to type another key quickly, which causes the `curses` program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the ESCAPE key.

Many existing programs use ESCAPE as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your `curses` programs, avoid the ESCAPE key.

`editor` — a Sample Program Listing

```
/* editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

#include <stdio.h>
#include <curses.h>

#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
        int i, n, l;
        int c;
        int line = 0;
        FILE *fd;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
                if (line > LINES - 2)
                        break;
                addch(c);
        }

        fclose(fd);

        move(0,0);
        refresh();
        edit();
```

```
        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
                n = len(l);
                for (i = 0; i < n; i++)
                        putc(mvinch(l, i) & A_CHARTEXT, fd);
                putc('\n', fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS - 1;

        while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;)

        {
                move(row, col);
                refresh();
                c = getch();

                /* Editor commands */
                switch (c)
                {

                /* hjkl and arrow keys: move cursor
                 * in direction indicated */
                case 'h':
                case KEY_LEFT:
                        if (col > 0)
                                col--;
                        else
                                flash();
                        break;

                case 'j':
                case KEY_DOWN:
                        if (row < LINES - 1)
                                row++;
                        else
                                flash();
                        break;

                case 'k':
                case KEY_UP:
```

```
                        if (row > 0)
                                row--;
                        else
                                flash();
                        break;
                case 'l':
                case KEY_RIGHT:
                        if (col < COLS - 1)
                                col++;
                        else
                                flash();
                        break;
                /* i: enter input mode */
                case KEY_IC:
                case 'i':
                        input();
                        break;
                /* x: delete current character */
                case KEY_DC:
                case 'x':
                        delch();
                        break;
                /* o: open up a new line and enter input mode */
                case KEY_IL:
                case 'o':
                        move(++row, col = 0);
                        insertln();
                        input();
                        break;
                /* d: delete current line */
                case KEY_DL:
                case 'd':
                        deleteln();
                        break;

                /* ^L: redraw screen */
                case KEY_CLEAR:
                case CTRL('L'):
                        wrefresh(curscr);
                        break;
                /* w: write and quit */
                case 'w':
                        return;
                /* q: quit without writing */
                case 'q':
                        endwin();
                        exit(2);
                default:
                        flash();
                        break;
                }
        }
}
/*
```

```
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

The `highlight` Program

This program illustrates a use of the routine `attrset()`. `highlight` reads a text file and uses embedded escape sequences to control attributes. `\U` turns on underlining, `\B` turns on bold, and `\N` restores the default output attributes.

Note the first call to `scrollok()`, a routine that we have not previously discussed (see `curses`(3V)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, `scrollok()` automatically scrolls the terminal up a line and calls `refresh()`.

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
```

```
        {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1], "r");

        if (fd == NULL)

        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr, TRUE);
        nonl();
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\\')
                {
                        c2 = getc(fd);
                        switch (c2)
                        {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }
                        addch(c);
                        addch(c2);
                }
                else
                        addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

The `scatter` Program

This program takes the first `LINES - 1` lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *      The scatter program.
 */

#include      <curses.h>
#include      <sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS]; /* Screen Array */
int  T[MAXLINES][MAXCOLS]; /* Tag Array - Keeps track of   *
                            * the number of characters     *
                            * printed and their positions. */

main()
{
        register int row = 0,col = 0;
        register int c;
        int char_count = 0;
        time_t t;
        void exit(), srand();

        initscr();
        for(row = 0;row < MAXLINES;row++)
                for(col = 0;col < MAXCOLS;col++)
                        s[row][col]=' ';

        col = row = 0;
        /* Read screen in */
        while ((c=getchar()) != EOF && row < LINES ) {

                if(c != '\n')

                {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if(c != ' ')
                                char_count++;
                }
                else
                {
                        col = 0;
                        row++;
                }
        }

        time(&t);     /* Seed the random number generator */
        srand((unsigned)t);

        while (char_count)
        {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (T[row][col] != 1 && s[row][col] != ' ')
```

```
                {
                        move(row, col);
                        addch(s[row][col]);
                        T[row][col] = 1;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

The `show` Program

`show` pages through a file, showing one screen of its contents each time you depress the space bar. The program calls `cbreak()` so that you can depress the space bar without having to hit return; it calls `noecho()` to prevent the space from echoing on the screen. The `nonl()` routine, which we have not previously discussed, is called to enable more cursor optimization. The `idlok()` routine, which we also have not discussed, is called to allow insert and delete line. (See `curses`(3V) for more information about these routines). Also notice that `clrtoeol()` and `clrtobot()` are called.

By creating an input file for `show` made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a `curses()` program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {

                perror(argv[1]);
                exit(2);
        }

        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
```

```
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw("%s", linebuf);
                }
                refresh();
                if (getch() == 'q')
                        done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

The `two` Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

`two` is just a simple example of a two-terminal `curses` program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command "`sleep 100000`" must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
        void done(), exit();
        unsigned sleep();
```

```
        char *getenv();
        int c;

        if (argc != 4)
        {
             fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
             exit(1);
        fd = fopen(argv[3], "r");
        fdyou = fopen(argv[1], "w+");
        signal(SIGINT, done);      /* die gracefully */

        me = newterm(getenv("TERM"), stdout, stdin);  /* initialize my tty */
        you = newterm(argv[2], fdyou, fdyou);        /* Initialize the other terminal */

        set_term(me);      /* Set modes for my terminal */
        noecho();          /* turn off tty echo */
        cbreak();          /* enter cbreak mode */
        nonl();            /* Allow linefeed */
        nodelay(stdscr, TRUE);     /* No hang on input */

        set_term(you);     /* Set modes for other terminal */
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);

        /* Dump second screen full on the other terminal */
        dump_page(you);

        for (;;) /* for each screen full */
        {
             set_term(me);
             c = getch();
             if (c == 'q')          /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(me);

             set_term(you);
             c = getch();
             if (c == 'q')          /* wait for user to read it */
             done();
             if (c == ' ')
             dump_page(you);
             sleep(1);
        }
}

dump_page(term)
  SCREEN *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line = 0; line < LINES - 1; line++) {
             if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
             clrtobot();
             done();
             }
```

```
                mvaddstr(line, 0, linebuf);
        }
        standout();
        mvprintw(LINES - 1, 0, "--More--");
        standend();
        refresh();          /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES - 1,0);          /* to lower left corner */
        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */
        /* Clean up second terminal */
        set_term(me);
        move(LINES - 1,0);          /* to lower left corner */
        clrtoeol();         /* clear bottom line */
        refresh();          /* flush out everything */
        endwin();           /* curses cleanup */
        exit(0);
}
```

The `window` Program

This example program demonstrates the use of multiple windows. The main display is kept in `stdscr`. When you want to put something other than what is in `stdscr` on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to `wrefresh()` for that window causes it to be written over the `stdscr` image on the terminal screen. Calling `refresh()` on `stdscr` results in the original window being redrawn on the screen. Note the calls to the `touchwin()` routine (which we have not discussed — see `curses`(3V)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a `curses` program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call `touchwin()` for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{
        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
```

```
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
                mvprintw(i, 0, "This is line %d of stdscr", i);

        for (;;)

        {
                refresh();
                c = getch();
                switch (c)

                {

                case 'c':     /* Enter command from keyboard */
                        werase(cmdwin);
                        wprintw(cmdwin, "Enter command:");
                        wmove(cmdwin, 2, 0);
                        for (i = 0; i < COLS; i++)
                                waddch(cmdwin, '-');
                        wmove(cmdwin, 1, 0);
                        touchwin(cmdwin);
                        wrefresh(cmdwin);
                        wgetstr(cmdwin, buf);
                        touchwin(stdscr);

                        /*
                         * The command is now in buf.
                         * It should be processed here.
                         */

                case 'q':
                        endwin();
                        exit(0);
                }

        }
}
```

A

make Enhancements Summary

A.1. New Features

Default Makefile

make's implicit rules and macro definitions are no longer hard-coded within the program itself. They are now contained in the default makefile /usr/include/make/default.mk. make reads this file automatically, unless there is a file in the local directory named default.mk. When you use a local default.mk file, you must add a directive to include the standard default.mk file to get the standard implicit rules and predefined macros.

The State File .make.state

make also reads a state file, .make.state in the directory. When the special-function target .KEEP_STATE is used in the makefile, make writes out a cumulative report for each target containing a list of hidden dependencies (as reported by compilation processors such as cpp), and the most recent rule used to build each target. The state file is very similar in format to an ordinary makefile.

Hidden Dependency Checking

When activated by the presence of the .KEEP_STATE target, make uses information reported from cc, cpp, f77, ld, make, pc and other compilation commands, and performs a dependency check against any header files (or in some cases, libraries) that are incorporated into the target file. These "hidden" dependency files do not appear in the dependency list, and often do not reside in the local directory.

Command Dependency Checking

When .KEEP_STATE is in effect, if any command line used to build a target changes between make runs (either as a result of editing the makefile or because of a different macro expansion), the target is treated as if it were out of date; make rebuilds it (even if it is newer that the files it depends on).

Automatic Retrieval of SCCS Files

Tilde Rules Superceded

This version of make automatically runs sccs get, as appropriate, when there is no rule to build a target file. A tilde appended to a suffix in the suffixes list indicates that sccs extraction is appropriate for dependency file. There are no longer special versions of implicit rules that include commands to extract current versions of sccs files.

To inhibit or alter the procedure for automatic extraction of the current `sccs` version, redefine the `.SCCS_GET` special-function target. An empty rule for this target inhibits automatic extraction entirely.

SCCS History Files

This version of `make` does not search the current directory for SCCS history (`s.`) files. These files must now reside in an `SCCS` subdirectory for `make`'s automatic version retrieval.

Pattern-Matching Rules: More Convenient than Suffix Rules

Pattern-matching rules have been added to simplify the process of adding new implicit rules of your own design. A target entry of the form:

tp`%`*ts* : *dp*`%`*ds*
rule

defines a pattern-matching rule for building a target from a related dependency file. *tp* is the target's prefix; *ts*, its suffix. *dp* is the dependency's prefix; *ds*, its suffix. The `%` symbol is a wild card that matches a contiguous string of zero or more characters appearing in both the target and the dependency filename. For example, the following target entry defines a pattern-matching rule for building a `troff` output file, with a name ending in `.tr` from a file that uses the `-ms` macro package ending in `.ms`:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

With this entry in the makefile, the command:

```
make doc.tr
```

produces:

```
hermes% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Using that same entry, if there is a file named `doc2.ms` the command:

```
make doc2.tr
```

produces:

```
hermes% make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

An explicit target entry overrides any pattern-matching rule that might apply to a target. Pattern-matching rules, in turn, normally override implicit rules. An exception to this is when the pattern matching rule has no commands in the rule portion of its target entry. In this case, `make` continues the search for a rule to build the target, and using as its dependency the file that matched the (dependency) pattern.

Pattern Replacement Macro References

As with suffix rules and pattern-matching rules, pattern replacement macro references has been added to provide a more general method for altering the values of words in a specific macro reference than that already provided by suffix replacement in macro references. A pattern-replacement macro reference takes the form:

$ (*macro* :*p* %*s*=*np* %*ns*)

where *p* is an existing prefix (if any), *s* is an existing suffix (if any), *np* and *ns* are the new prefix and suffix, respectively, and % is a wild card character matching a string of zero or more characters within a word. The prefix and suffix replacements are applied to all words in the macro value that match the existing pattern. Among other things, this feature is useful for prefixing the name of a subdirectory to each item in a list of files. For instance, the following makefile:

```
SOURCES= x.c y.c z.c
SUBFILES.o= $(SOURCES:%.c=subdir/%.o)

all:
        @echo $(SUBFILES.o)
```

produces:

```
hermes% make
subdir/x.o subdir/y.o subdir/z.o
```

You may use any number of % wild cards in the right-hand (replacement) side of the equal-sign, as needed. The following replacement:

```
...
NEW_OBJS= $(SOURCES:%.c=%/%.o)
```

would produce:

```
...
x/x.o y/y.o z/z.o
```

Please note, however, that pattern-replacement macro references

Please note that pattern-replacement macro references should not appear on the dependency line of a pattern-matching rule's target entry. This produces unexpected results. With the makefile:

```
OBJECT= .o

x:
%: %.$(OBJECT:%o=%Z)
        cp $< $@
```

it looks as if `make` should attempt to build a target named, `x` from a file named `x.Z`. However, the pattern-matching rule is not recognized; `make` cannot determine which of the `%` characters in the dependency line apply to the pattern-matching rule, and which apply to the macro reference. Consequently, the target entry for `x.Z` is never reached. To avoid problems like this, you can use an intermediate macro on another line:

```
OBJECT= .o
ZMAC= $(OBJECT:%o=%Z)

x:
%: %$(ZMAC)
        cp $< $@
```

New Options

There are a number of new options:

`-d` Display dependency-check results for each target processed. Displays all dependencies that are newer, or indicates that the target was built as the result of a command dependency.

`-dd` The same function as `-d` had in earlier versions of `make`. Displays a great deal of output about all details of the `make` run, including internal states, etc.

`-D` Display the text of the makefile as it is read.

`-DD` Display the text of the makefile, and of the default makefile being used.

`-p` Print macro definitions and target entries.

`-P` Report all dependencies for targets without rebuilding them.

Support for C++ and Modula-2

This version of `make` contains predefined macros for compiling C++ programs. It also contains predefined macros and implicit rules for compiling Modula-2.

Naming Scheme for Predefined Macros

The naming scheme for predefined macros has been rationalized, and the implicit rules have been rewritten to reflect the new scheme. The macros and implicit rules are upward compatible with existing makefiles.

For example, there is now a macro called `SUFFIXES`, that contains the default entries for the suffixes list; the target entry for the default suffixes list looks like:

```
.SUFFIXES: $(SUFFIXES)
```

If you want to insert new suffixes at the head of the list, you can do so quite simply as follows:

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Other examples include the macros for standard compilations commands:

`LINK.c` Standard `cc` command line for producing executable files.

`COMPILE.c` Standard `cc` command line for producing object files.

New Special-Purpose Targets

The `.KEEP_STATE` target should not be removed once it has been used in a `make` run.

`.KEEP_STATE` When included in a makefile, this target enables hidden dependency and command dependency checking. In addition, `make` updates the state file `.make.state` after each run.

`.INIT` and `.DONE`
These targets can be used to supply commands to perform at the beginning and end, respectively, of each `make` run.

`.FAILED` The commands supplied are performed when `make` fails.

New Implicit Rule for `lint`

Implicit rules have been added to support incremental verification with `lint`.

Macro Processing Changes

A macro's value can now be of virtually any length. Whereas in earlier versions only trailing white space was stripped from a macro's value, this version strips off both leading and trailing white space characters.

Macros: Definition, Substitution, and Suffix Replacement

New Append Operator: +=
This operator appends a SPACE, followed by a word or words, onto the existing value of the macro.

Conditional Macro Definitions: :=
This operator indicates a conditional (targetwise) macro definition. A makefile entry of the form:

target := *macro* = *value*

indicates that *macro* takes the indicated *value* while processing *target* and its dependencies.

Patterns in Conditional Macros

`make` recognizes the `%` wild card pattern in the target portion of a conditional macro definition. For instance:

```
profile_% := CFLAGS += -pg
```

would modify the `CFLAGS` macro for all targets having the '`profile_`' prefix. Pattern replacements can be used within the value of a conditional definition. For instance:

```
profile_% := OBJECTS = $(SOURCES:%.c=profile_%.o)
```

will apply the `profile_` prefix and `.o` suffix to the basename of every `.c` file in the `SOURCES` list (value).

Suffix Replacement Precedence
Substring replacement now takes place following expansion of the macro being referred to. Previous versions of `make` applied the substitution first, with results that were counterintuitive.

Nested Macro References
`make` now expands inner references before parsing the outer

reference. So, a nested reference as in this example:

```
CFLAGS-g = -I../include
OPTION = -g
$(CFLAGS$(OPTION))
```

now yields the value `-I../include`, rather than a null value, as it would have in previous versions.

Cross-Compilation Macros

The predefined macros `HOST_ARCH`, `HOST_MACH`, `TARGET_ARCH`, and `TARGET_MACH` are available for use in cross-compilations. By default, the *arch* macros are set to the value returned by the `arch` command; the *mach* macros are set to the value returned by `mach`.

Shell Command Output in Macros

A definition of the form:

MACRO `:sh =` *command*

sets the value of *MACRO* to the standard output of the indicated *command*, [NEWLINE] characters being replaced with [SPACE] characters. The command is performed just once, when the definition is read. Standard error output is ignored, and `make` halts with an error if the command returns a non-zero exit status.

A macro reference of the form:

`$(`*MACRO* `:sh)`

expands to the output of the command-line stored in the value of *MACRO*, whenever the reference is evaluated. [NEWLINE] characters are replaced with [SPACE] characters, standard error output is ignored, and `make` halts with an error if the command returns a non-zero exit status.

Improved `ar` Library Support

Lists of Members

`make` automatically updates an `ar`-format library member from a file having the same name as the member. Also, `make` now supports lists of members as dependency names of the form:

lib.a: *lib.a*(*member member* ...)

Handling of `ar`'s Name Length Limitation

make now copes with the 15-character member-name length limitation in `ar`. It now recognizes a member name that matches the first 15 characters of a filename as the member corresponding to the file.

Target Groups

It is now possible to specify that a rule produces a set of target files. A + sign between target names in the target entry indicates that the named targets comprise a group. The target group's rule is performed once, at most, in a `make` invocation.

A.2. Incompatibilities with Previous Versions of make

New Meaning for -d Option

The –d option now reports the reason why a target is considered out of date.

Dynamic Macros

Although the dynamic macros $< and $* were documented being assigned only for implicit rules and the .DEFAULT target, in some cases they actually were assigned for explicit target entries. The assignment action is now documented properly.

The actual value assigned to each of these macros is derived by the same procedure used within implicit rules (this hasn't changed). This can lead to unexpected results when they are used in explicit target entries.

Even if you supply explicit dependencies, make doesn't use them to derive values for these macros. Instead, it searches for an appropriate implicit rule and dependency file. For instance, if you have the explicit target entry:

```
test: test.f
        @echo $<
```

and the files: test.c and test.f, you might expect that $< would be assigned the value test.f. This is *not* the case. It is assigned test.**c**, because .c is ahead of .f in the suffixes list:

```
hermes% make test
test.c
```

For explicit entries, we recommend a strictly deterministic method for deriving a dependency name using macro references and suffix replacements. For example, you could use: $@.f instead of $< to derive the dependency name. To derive the basename of a .o target file, you could use the suffix replacement macro reference: $(@:.o=) instead of $*.

When hidden dependency checking is in effect, the $? dynamic macro's value includes the names of hidden dependencies, such as header files. This can lead to failed compilations when using a target entry such as:

```
x: x.c
        $(LINK.c) -o $@ $?
```

and the file x.c #include's header files. The workaround is to replace '$?' with '$@.<'.

Tilde Rules not Supported

This version of make does not support tilde suffix rules for version retrieval under SCCS. This may create problems when older makefiles redefine tilde rules to perform special steps when version retrieval under SCCS is required.

Target Names Beginning with ./ Treated as Local Filenames

When `make` encounters a target name beginning with '`./`', it strips those leading characters. For instance, the target named:

```
./filename:
```

is interpreted as if it were written:

```
filename:
```

This can result in endless loop conditions when used in a recursive target. To avoid this, rewrite the target relative to '`..`', the parent directory:

```
../dir/filename
```

Index

D

L

N

O

P

T

U

V

W

X

Y

Z

Notes

Notes

Notes

Notes

Notes