Compilation Systems Volume 1 (Tools)

CONCURRENT COMPUTER CORPORATION™

0890459-050

April 1999

Acknowledgment: This manual contains material contributed by 88open Consortium, Ltd.

In this document, the term 601 is used as an abbreviation for the phrase “PowerPC 601 RISC microprocessor.” The terms 603, 604, 620, and 640 are used similarly.

Revision History:	Level:	Effective With:
Original Release -- October 1994	000	PowerUX 1.0
Previous Release -- March 1997	040	PowerMAX OS 4.1
Current Release -- April 1999	050	PowerMAX OS 4.3

Preface

Scope of Manuals

The Compilation Systems Manual set is composed of two manuals: *Compilation Systems Volume 1 (Tools)* and *Compilation Systems Volume 2 (Concepts)*. The *Compilation Systems Volume 1 (Tools)* manual describes the features and use of several software development environment tools, analysis tools, and project-control tools. The *Compilation Systems Volume 2 (Concepts)* manual describes the concepts behind compilation systems including environments, performance analysis, and formats.

Information in this manual applies to the PowerPC™ platforms described in the *Concurrent Computer Corporation Product Catalog*.

Structure of Manuals

A brief description of the parts, chapters, and appendixes in the *Compilation Systems Volume 1 (Tools)* manual follows:

Part 1 discusses software development environment tools.

Chapter 1 introduces compilation system tools and concepts.

Chapter 2 describes the assembly language, and it discusses the assembler, **`as`**.

Chapter 3 summarizes the instructions, condition codes, operands, and registers associated with the PowerPC.

Chapter 4 covers the link editor, **`ld`**. It also discusses dynamic linking, plus the creation and use of shared objects.

Chapter 5 describes the macro processor, **`m4`**.

Chapter 6 presents the lexical analyzer, **`lex`**.

Chapter 7 presents the compiler-compiler, **`yacc`**.

Part 2 describes analysis tools.

Chapter 8 provides an introduction to the other chapters in this part.

Chapter 9 presents the C code browser, **`cscope`**.

Chapter 10 discusses the C code checker, **`lint`**.

Chapter 11 discusses performance analysis and use of the **`analyze`** and **`report`** utilities.

Part 3 presents project-control tools.

Chapter 12 provides an introduction to the other chapters in this part.

Chapter 13 presents the **make** utility.

Chapter 14 covers the **sccs** source code control system.

A brief description of the parts, chapters, and appendixes in the *Compilation Systems Volume 2 (Concepts)* manual follows:

Part 4 discusses environments.

Chapter 15 provides an introduction to the other chapters in this part.

Chapter 16 provides an overview of commonly-used system libraries.

Chapter 17 discusses the IEEE floating-point operations used on supporting hardware platforms.

Chapter 18 describes interfaces between C and Fortran routines on supporting hardware platforms.

Part 5 describes performance analysis concepts.

Chapter 19 provides an introduction to the other chapters in this part.

Chapter 20 provides a tutorial on program optimization, focusing on the optimizations performed by the Concurrent compilers.

Part 6 covers formats.

Chapter 21 provides an introduction to the other chapters in this part.

Chapter 22 describes the executable and linking format, ELF.

Chapter 23 discusses text description information, tdesc.

Chapter 24 describes the debugging information format, DWARF. It is primarily a reprint of the DWARF specification from UNIX International.

Chapter 25 covers the libdwarf library that provides access to DWARF debugging and line number information. It is primarily a reprint of a document from UNIX International.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic.*
list bold	User input appears in **list bold** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **list bold** type.

`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type. Keywords also appear in `list` type.
<u>emphasis</u>	Words or phrases that require extra emphasis use <u>emphasis</u> type.
`window`	Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in `window` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.
`{}`	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
`...`	An ellipsis follows an item that can be repeated.

The window images in this manual come from a Motif environment. If you are using another environment, your windows may differ slightly from those presented here.

Referenced Publications

The following publications are referenced in this document:

0890240	hf77 Fortran Reference Manual
0890288	HAPSE Reference Manual
0890395	NightView User's Guide
0890398	NightTrace Manual
0891019	Concurrent C Reference Manual

The vendor publications referenced in this manual may be viewed on the respective's companies WWW site.

Contents

Tables

1
Software Development Environments

Replace with Part 1 tab

Part 1 - Software Development Environments

1
Introduction to SDEs

1 Introduction to SDEs

Introduction

To create a program, you must be working in and understand some aspects of a software development environment (SDE). A *software development environment* includes the hardware, operating system, supported object and debugging information formats, compilers and utilities.

This part of the manual discusses some of the tools available in the software development environment.

Chapter 2 ("Assembler and Assembly Language") covers the instruction mnemonics and assembler implementation for the supporting hardware platforms[1].

Chapter 3 ("PowerPC Instruction Set Summary") summarizes the instructions, condition codes, operands, and registers associated with the PowerPC.

Chapter 4 ("Link Editor and Linking") describes the **`ld`** link editor and static and dynamic linking of relocatable object files and libraries (including relocatable archives and shared objects). For information about compressing common object files, see **`cprs(1)`**.

Chapter 5 ("m4 Macro Processor") discusses preprocessing C, RATFOR, assembly language, and other source files with built-in and user-defined **`m4`** macros. For information about the C preprocessor, see **`cpp(1)`** and **`acpp(1)`**.

Chapter 6 ("Lexical Analysis with lex") describes how to write specifications for **`lex`** to separate (and possibly generate statistics for) components of program input.

Chapter 7 ("Parsing with yacc") explains how to write grammar rules for **`yacc`** so that it can act upon identified components of program input.

The following sections describe compilation systems.

Programming Languages

Programming languages are used for specifying instructions and operations which are to be performed by programs running on a computer system. Like the spoken languages that all human beings use, each programming language has a grammar and a set of syntactic and semantic rules.

1. See the Preface for details.

There are hundreds of programming languages available to the computing world. Concurrent Computer Corporation supports a few of the most popular languages:

C	See the Concurrent *C Reference Manual*
Fortran	See the *hf77 Fortran Reference Manual*
Ada	See the *HAPSE Reference Manual*
assembly language	See Chapter 2 (“Assembler and Assembly Language”) in this manual.

C, Fortran, and Ada are often referred to as *high-level languages*. The source code for programs written in these languages is fairly portable across computer systems provided by different manufacturers. In addition, these programs can be accepted and processed by compilers produced by different software vendors. The literary world provides an abundance of books and references on these languages.

Assembly language is often referred to as a *low-level language*. This language provides mnemonics and directives which usually map one-to-one with the instruction set and resources of the computer system.

All of these languages are supported for the supporting hardware platforms.

Compilation Systems Concepts

A *compilation system* is a set of language processors, commands, utilities, and libraries which can be used in the development of software programs. Compilation systems convert source code into binary programs which can be executed on a computer. In addition, they provide tools and facilities for debugging and analyzing program behavior and characteristics.

At the heart of a compilation system is the *language processor*. Usually, this is the *compiler*. A compiler is a program which accepts, as input, source code written in a high-level language. It processes this input and produces a lower-level representation of the source code. This new representation can be an assembly language representation of the higher-level source code, making it necessary to run an assembler to produce a machine-level representation of the code. Sometimes a compiler will translate the high-level language directly into the machine-level representation. A compiler analyzes the source code, both syntactically and semantically. A good compiler detects as many errors as it can locate, enabling the programmer to correct them before they occur during execution of the program. A good compiler can also *optimize* the program. Optimization transforms the program, allowing it to run faster and more efficiently.

Some languages are processed by an *interpreter*. Whereas a compiler produces output that must be further processed and then executed, an interpreter performs “on the fly” translation and execution of the program.

Assembly language is processed by an *assembler*. Assemblers are usually less sophisticated than compilers and interpreters. An assembler often does nothing more than convert the specified assembly language instructions and directives into machine instructions.

Compilers and assemblers are used to produce *relocatable object files*. Each of these files cannot be executed individually, for they require further processing. An *executable program* consists of one or more relocatable object files. It is produced by a *link editor*. One relocatable object file may reference routines and/or data that are provided by another relocatable object file. The link editor resolves these references.

Sometimes it is useful to maintain a library of relocatable object files. A programmer could then include object files from the library with object files that are specific to the program. This *library* of relocatable object files is also referred to as an *archive*, and the archive is maintained by an *archiver*. When utilizing an archive, the link editor incorporates into the program only those relocatable object files which are needed by the program. The system on which a program is developed provides several system libraries.

Newly-written programs seldom execute correctly on the first run, requiring the programmer to debug the program. A *debugger* utility is often used to facilitate the search for problems in the code. Some debuggers operate only at the level of machine instructions. *Symbolic debuggers* permit debugging at the source code level.

Once a program is running correctly, it is sometimes desirable to analyze its performance and identify bottlenecks during its execution. *Profiling* tools are called upon to perform this analysis. These tools are available in various degrees of complexity.

Finally, compilation systems provide a set of tools for examining, compressing, and performing miscellaneous functions on source code, relocatable object files, and executable programs.

Concurrent Computer Corporation Compilation Systems

Concurrent Computer Corporation’s compilation systems provide all of the facilities described above, except for interpreters. The compilers produce assembly language files. These are assembled into relocatable object files, and the relocatable object files are combined into an executable program.

Concurrent Computer Corporation’s compilers share a common *back end* which is responsible for optimization and code generation. This Common Code Generator (CCG) technology makes it possible to easily add support for new languages and to retarget existing compilers to new hardware platforms.

Concurrent Computer Corporation has developed its own compilers. They are not reincarnations of compilers produced by other vendors.

Table 1-1 shows which compilers and utilities are available.

Table 1-1. Compilers and Utilities

Type	Name	Description
C compiler	**cc(1)**	Both ANSI C and “old-style” C are accepted, as are Concurrent Computer Corporation extensions to the C language.
Fortran compiler	**f77(1)**	The ANSI Fortran 77 language is accepted, as are Concurrent Computer Corporation extensions to the Fortran language.
Ada compiler	**ada(1)**	Concurrent Computer Corporation provides a complete Ada programming support environment known as HAPSE.
C preprocessors	**cpp(1)** **acpp(1)**	The C preprocessor expands macros and performs other preprocessing functions on the source code as part of the compilation.
Assembler	**as(1)**	Each system supported by Concurrent Computer Corporation uses a “base” assembly language that is supported by other vendors of the underlying architecture. Extensions are added to this language.
Link editor	**ld(1)**	The Concurrent link editor produces programs which can use either static linking or dynamic linking.
Archiver	**ar(1)**	The Concurrent archiver is optimized for fast archive operations.
Post-link optimizer and profiler	**analyze(1)** **report(1)**	These tools are unique to Concurrent **analyze(1)** can be used to perform additional optimizations on programs that have been link edited. It can also be used to obtained profiling and timing information for executable programs. **report(1)** provides readable profiling data.
Profiler	**prof(1)**	This tool is the standard UNIX® profiling utility. It is available but not useful on the supporting hardware platforms.
Performance analyzer	**NightTrace(1)**	This tool is unique to and can be purchased from Concurrent Computer Corporation. It allows users to analyze data and timings in user applications and the kernel. See the *NightTrace Manual* for details.
Symbolic debugger	**gdb(1)**	This is a port of the Free Software Foundation’s GNU debugger. Concurrent has added support for the Fortran language and for DWARF symbolic debugging information.
Symbolic debugger	**NightView(1)**	This source-level, multi-lingual, multi-process debugger is unique to and can be purchased from Concurrent Computer Corporation. See the *NightView User’s Guide* for details.
Symbolic debugger	**ctrace(1)**	This utility displays source statements as they execute. It also shows variable names and values and any output from the statement.
Object debugger	**adb(1)**	This debugger, provided on some vendors’ UNIX systems, allows a program to be debugged at the instruction level.
Compiler-compiler	**yacc(1)**	This utility converts a context-free grammar into a set of tables for a simple automation which uses an LR(1) parsing algorithm.
Lexical analyzer	**lex(1)**	This utility generates simple code to be used in the lexical analysis of text input.

Table 1-1. Compilers and Utilities (Cont.)

Type	Name	Description
C code checker	**`lint(1)`**	This utility examines C source for syntax errors and incompatible routine interfaces.
C code browser	**`cscope(1)`**	This utility is used for browsing C source code for specified elements.
C cross reference generator	**`cxref(1)`**	This utility builds a cross reference table from C source files.
Name lister	**`nm(1)`**	This utility is used to provide a readable display of an object file's symbol table.
Section manipulator	**`mcs(1)`**	This utility adds, deletes, prints, or compresses a section, by default the **`.comment`** section, in an ELF object file.
Dumper	**`dump(1)`**	This utility is used to provide a readable display of all components of an object file.
Sizer	**`size(1)`**	This utility gives the byte size of selected sections of an object file.
Stripper	**`strip(1)`**	This utility is used to remove the symbol table from an object file.
Compressor	**`cprs(1)`**	This utility, available on some UNIX systems for compression of COFF symbolic debug information in an object file, has been adapted by Concurrent Computer Corporation to compress DWARF symbolic debug information from ELF files.
Disassembler	**`dis(1)`**	This utility provides a readable display of the machine level instructions in an object file.
pc to line number and file name translator	**`pctolf(1)`**	This utility is unique to Concurrent Computer Corporation. For a particular program counter value within an object file, it utilizes DWARF symbolic debug information to present the file name and line number which correspond to that address.
Macro preprocessor	**`m4(1)`**	This utility serves as a macro processor front end for source files written in C and other languages.
Ordering identifier	**`lorder(1)`**	This utility finds the ordering relation of object files for a library.
C flow grapher	**`cflow(1)`**	This utility builds a graph of external function references from C, **`yacc`**, **`lex`**, assembler, and object files.
Topological sorter	**`tsort(1)`**	This utility provides an ordered list of items, which are usually the output from **`lorder(1)`**.

Object Files

An *object file* is a binary container of machine instructions and reference information. Relocatable object files and executable programs are two kinds of object files.

An object file must have a well-defined format if it is to be used by the various utilities in a compilation system. The object file format used under PowerUX is the *Executable and Linking Format (ELF)*. This format provides object file sections, which contain the various components of an object file, such as the machine-level instructions, relocation information, and the symbol table. It also specifies the segments an executing program will have in the address space.

Information about an object file that can be used by a symbolic debugger is often embedded within the object file. ELF was designed to be independent of any particular representation of symbolic debugging information. Thus, *Debugging With Arbitrary Record Format (DWARF)* has become the de facto representation for use with ELF, and it is used under PowerUX.

Stack Frames

During execution, a computer program utilizes a portion of its address space known as the *stack*. Each subroutine or procedure that is currently active utilizes a contiguous group of words on the stack, which is that subroutine's *stack frame*. The stack frame contains such information as the address to which the subroutine should return when it completes its execution, the address of the stack frame corresponding to the subroutine which invoked the current subroutine, the values of certain registers upon entry to the current subroutine, and the values of data variables visible only to the current subroutine.

Some computer architectures provide hardware support for stack frames. Modern architectures have made the stack frame a software concept, leaving control of stack frames to the executable program. Compilers, then, generate code which causes each subroutine to create, update, and remove its own stack frame.

The absence of hardware support for stack frames would make it virtually impossible for a debugger to produce a stack traceback, which is an identification of the invocation order of subroutines at any point in time during execution of the program. Concurrent compilation systems are able to support stack tracebacks through the use of *text description information*, or tdesc. This information, embedded within an executable program, describes pertinent portions of subroutines to the debugger.

Static and Dynamic Linking

Programs may be developed under PowerUX with static linking or dynamic linking. A *statically linked* program contains all of the code and data it will need during execution. The link editor supplies the program with these necessary components.

A *dynamically linked* program does not contain all of the code and data it will need during execution. The link editor statically links a portion of the code and data into the executable program. When the program begins execution, a *program interpreter* dynamically links into the executing program's process' address space the remaining code and data needed by the program. This additional code and data is provided by *shared objects*, or *shared libraries*.

Dynamically linked programs provide greater sharing of pages of memory, and their on-disk images are smaller than those of equivalent statically linked programs. Statically linked programs, however, typically run faster than dynamically linked programs.

Floating-Point Arithmetic

The representation and operations of floating-point numbers varies among computer systems. The supporting hardware platforms uses the representation and operations specified by the *IEEE Standard for Binary Floating-Point Arithmetic*, which has become a de facto standard for floating-point arithmetic.

Concurrent compilation systems support the single precision and the double precision formats. No support is provided for the double extended precision format.

2
Assembler and Assembly Language

2
Assembler and Assembly Language

Introduction

Concurrent Computer Corporation’s assembler is available on supporting hardware platforms[1]. The assembler accepts instruction mnemonics appropriate to the particular underlying architecture. An extended set of directives, or pseudo-ops, extends the programmer’s capability for specifying data and section control. A subset of these directives is common to each platform.

The following sections describe the assembly statements and directives. The available instructions and their syntax and semantics may be found in the reference manuals and documents listed below.

Title
Chapter 3 (“PowerPC Instruction Set Summary”) of this manual
Assembler Language Reference for IBM® AIX™ Version 3 for RISC System/6000™
PowerPC 604 RISC Microprocessor User’s Manual
PowerPC Microprocessor Family: The Programming Environments
PowerPC User Instruction Set Architecture

Assembler Operation

Input to the assembler is a source file containing instruction mnemonics and directives. The assembler processes this input in two passes. During the first pass, it reads each of the instructions and directives, creates a symbol table containing information about every symbol seen within the assembly source, and creates other internal tables describing the instructions and directives it reads. During the second pass, the assembler creates a relocatable object file. This object file is in ELF format. (See Chapter 22 (“Executable and Linking Format (ELF)”) for details.) The **`.text`** section of the object file contains the binary encodings of the assembly instructions in the source. Historically, this collection of bits and bytes has been referred to as *machine language*. The **`.data`** and the **`.bss`**

1. See the Preface for details.

sections contain the initialized and uninitialized data, respectively. The **.symtab** section contains information about all of the symbols present in the assembly source. The **.rela_*** sections provide relocation information to the link editor, enabling it to combine this relocatable object file with other such files to form an executable program.

The assembler processes only one input file on each invocation. Traditionally, the name of an input file ends with the suffix **.s**, although any valid UNIX name is acceptable. The **-o** option can be used to specify the name of the output object file. If this option is not used, the assembler names the output file according to the following rules:

- If the name of the input file ends in **.s**, then the name of the output file is the same as the name of the input file, but with **.s** replaced with **.o**.
- If the name of the input file does not end in **.s**, then the name of the output file is the same as the name of the input file, but with **.o** appended.

The C, Fortran, and Ada compilers produce assembly language source file(s) as their compiled output. They then invoke the assembler to convert the assembly source files into relocatable object files.

Temporary files are used during assembly. If the `TMPDIR` environment variable is defined, these files are placed under this directory. If it is not defined, the **/var/tmp** directory is used, if it is available; otherwise, **/tmp** is used. Temporary files are removed by the assembler upon completion of assembly.

Using the Assembler

Assembler Invocation

The assembler is invoked as:

as [*options*] *file*

The options are listed below.

-f *float* — Use *float* as the floating-point mode of assembly and the object file.

Desired Mode	Acceptable Argument Values			
IEEE-COMPATIBLE	3	ieeecom		
IEEE-NEAREST	4	ieeenear	near	ieee
IEEE-ZERO	5	ieeezero	zero	
IEEE-POS-INFINITY	6	ieeepos	pos	
IEEE-NEG-INFINITY	7	ieeeneg	neg	

-m — Run the **m4** macro preprocessor on the input to the assembler.

-o *objfile* — Put the output of the assembly in *objfile* by default. The output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

-A — Accept certain extensions to the Ada language.

(1) Allow a string enclosed in double quotes to appear in an identifier, provided the first character of the identifier is not a double quote. The characters normally allowed in an identifier may appear in the quoted string. Additionally, the characters `+, -, *, /, =, <, >`, and & may appear in the quoted string.

(2) Allow multiple file directives in the source program.

-P — Create a formatted listing on standard output. The format of the printout is:

’*line-number pc memory-layout source-line*’

The ’*pc*’ field in a **.data** section will be followed by a ’`*`’.

-QTARGET=PPC601 — Mark the object module as using features unique to the PowerPC 601, and provide warnings for any assembly instructions which are unique to another PowerPC chip architecture.

-QTARGET=PPC603 — Mark the object module as using features unique to the PowerPC 603, and provide warnings for any assembly instructions which are unique to another PowerPC chip architecture.

-QTARGET=PPC604 — Mark the object module as using features unique to the PowerPC 604, and provide warnings for any assembly instructions which are unique to another PowerPC chip architecture.

-QTARGET=PPC604E — Mark the object module as using features unique to the PowerPC 604e, and provide warnings for any assembly instructions which are unique to another PowerPC chip architecture.

-QTARGET=PPC620 — Mark the object module as using features unique to the PowerPC 620, and provide warnings for any assembly instructions which are unique to another PowerPC chip architecture.

-QTARGET=PPCCOMPAT
Mark the object module as using only features common to all the PowerPC platforms, and provide warnings for any assembly instructions which are unique to any of the platforms.

-Q{**y**|**n**} — If **-Qy** is specified, place the version number of the assembler being run in the object file. The default is **-Qn**.

-R — Remove (unlink) the input file after assembly is completed.

-V — Write the version number of the assembler being run on the standard error output.

-Y [**md**],*dir* — Find the **m4** preprocessor (**m**) and/or the file of predefined macros (**d**) in directory *dir* instead of in the customary place.

Character Set

The standard ASCII characters and special two-character combinations comprise the assembly character set. When used in identifiers and labels, letters are case-sensitive. That is, the symbols `VAL25` and `val25` are distinct symbols. Letters are not case-sensitive in instruction and directives mnemonics. Thus `word` and `WORD` identify the same directive.

Source Statements

Source statements may appear on individual lines, or multiple statements may be specified on a single line separated by the `;`delimiting character.

Any statement may be preceded by one or more labels.

The assembler imposes no limit on the character length of a source line.

Null Statements

Null statements are empty lines or lines containing only one or more labels. Such statements are ignored by the assembler.

Alphanumeric Labels

Alphanumeric labels consist of the following characters:

```
a-z, A-Z, 0-9, _, ., $, %, and @
```

These labels must not begin with a digit.

If the assembler **`-A`** option is used, labels may also contain double-quoted strings of the preceding character set and the characters `+, -, *, /, =, <, >`, and &.

Labels may be preceded by zero or more blanks. They are terminated by a `:` (which does not become part of the label name). One or more blanks may precede the colon. The assembler does not prefix or suffix additional underscores to the label, as some compilers do. If a **`version "03.00"`** or a **`version "02.00"`** directive (discussed in “ELF Symbol Attributes” on page 2-17) does not exist in the assembly file, the assembler removes a leading underscore, if one exists, from label names. If a **`version "03.00"`** or a **`version "02.00"`** directive does exist in the assembly file, the assembler does not remove a leading underscore from labels. Alphanumeric labels have a maximum length of 1,024 characters. For example,

```
_label1:   PCB.flag:
```

An alphanumeric label assigns the current value and type of the location counter to the named symbol. In the **`.text`** section, the location counter is the program counter in that section. In other sections, the location counter is the address of the next data byte in that section.

Numeric (Local) Labels

A *numeric label* consists of a digit `0-9` followed by a colon. It defines a temporary symbol of the form *n*`b` or *n*`f`, where *n* is the digit of the label and `b` or `f` indicates a *backward* or *forward* reference, respectively. A numeric label assigns the current value and type of the location counter to the temporary symbol. For example, the operand `3b` refers to the nearest label `3:` seen prior to the instruction, and `8f` refers to the nearest label `8:` seen after the instruction.

For example,

```
6:

     cmpwi   crf1,r3,13
     addi    r3,r3,1
     bgt     crf1,6f
     cmpwi   crf1,r3,4
     blt     crf1,6b
6:
```

NOTE

The symbol `0f` may not be used as a numeric label because `0f` denotes the floating-point constant `0.0`.

Comments

A line with # in column 1 is regarded as a comment line.

C-style comments, beginning with `/*` and ending with `*/`, may appear anywhere in the source. These comments may traverse multiple lines.

Comments to the end of the line may also be used. The delimiter for this kind of comment is #. This delimiter can be used anywhere on the line.

Identifiers

Identifiers consist of the following characters:

```
a-z, A-Z, 0-9, _, ., $, %, and @
```

Identifiers must not begin with a digit.

If the assembler **`-A`** option is used, identifiers may also contain double-quoted strings of the preceding character set and the characters `+, -, *, /, =, <, >`, and &.

The assembler does not prefix or suffix additional underscores to the identifier, as some compilers do. If a **`version "03.00"`** or a **`version "02.00"`** directive (discussed "ELF Symbol Attributes" on page 2-17) does not exist in the assembly file, the assembler removes a leading underscore, if one exists, from identifiers. If a **`version "03.00"`** or a **`version "02.00"`** directive does exist in the assembly file, the assembler does not remove a leading underscore from identifiers. Identifiers have a maximum length of 1024 characters. Examples of identifiers include:

```
@L5,   _subroutine_
```

Each identifier (symbol) may be classified as either *predefined* by the assembler or *user-defined.*

Predefined Symbols

These symbols possess specific meanings for the assembler. They cannot be redefined by the user, nor may they be used outside their specific contexts.

Predefinitions:

- Instruction mnemonics
- Assembler directives (see "Assembler Directives" on page 2-12)
- General register names: `r0 - r31`
- Floating-point register names: `f0 - f31`
- Special-purpose register names: `xer, lr, ctr, dsisr, dar, dec, sdr1, srr0, srr1, sprg0, sprg1, sprg2, sprg3, ear, pvr, ibat0u, ibat0l, ibat1u, ibat1l, ibat2u, ibat2l, ibat3u, ibat3l, iabr`
- PowerPC 601-specific special-purpose register names: `mq, rtcu, rtcl, dec, hid1`
- PowerPC 603-specific special-purpose register names: `dmiss, dcmp, hash1, hash2, imiss, icmp, rpa`
- PowerPC 604-specific special-purpose register names: `mmcr0, pmc1, pmc2, sia, sda`
- PowerPC 620-specific special-purpose register names: `asr, mmcr0, pmc1, pmc2, sia, sda, buscsr, l2cr, l2sr, fpecr`
- Special-purpose register names absent from PowerPC 601: `tb, tbl, dbat0u, dbat0l, dbat1u, dbat1l, dbat2u, dbat2l, dbat3u, dbat3l`

- Special-purpose register names absent from PowerPC 603: `hid0, dabr, pir`
- Special-purpose register names absent from PowerPC 601 and 620: `tbu`
- Half-word specifiers:

`hi16`	(upper 16 bits of a relocatable expression, for signed operations)
`uhi16`	(upper 16 bits of a relocatable expression, for unsigned operations)
`lo16`	(lower 16 bits of a relocatable expression)

NOTE

An instruction which uses `lo16` as the half-word specifier often has a corresponding instruction which provides the upper 16 bits of a relocatable expression. If the instruction using `lo16` performs a sign extension of the 16-bit operand, then `hi16` should be used in the corresponding instruction which provides the upper 16 bits; otherwise, `uhi16` should be used in the corresponding instruction. For example:

```
lis   rs,uhi16(x)
ori   rs,rs,lo16(x)
```

but

```
lis   rs,hi16(x)
addi  rs,rs,lo16(x)
```

- Branch instruction operands: `eq, ne, gt, le, lt, ge, so, un, ns, z, nl, ng, nz, nu`

If the **`version "03.00"`** directive is specified in an assembly file, the assembler requires that a leading @ be prefixed to the following predefinitions:

- General register names
- Extended register names
- Control register names
- Half-word specifiers
- Bit-number mnemonics
- Match-field mnemonics

User-Defined Symbols

The user may define a symbol in one of the following ways.

- As a label. The symbol's value is the value of the location counter where the label is defined.
- As a constant. The **`def`** directive can be used to assign a 32-bit integer value to the symbol. (Refer to "Symbol Definitions" on page 2-16.)
- As a special symbol. The **`file`** directive, for example, can be used to give the symbol a special meaning. (Refer to "Miscellaneous Operations" on page 2-18 and "Summary of Directives Mnemonics" on page 2-19.)

Constants

Integer Constants

An *integer constant* is a 32-bit, two's complement number.

A *decimal constant* consists of digits from `0-9` and does not possess a leading zero.

An *octal constant* consists of digits from `0-7` and possesses a leading zero.

A *hexadecimal constant* consists of digits from `0-9`, `a-f`, and `A-F` and possesses a leading `0x` or `0X`. For example,

```
914, 037775, 0x23a
```

Floating-Point Constants

A *floating-point constant* is a 32-bit or a 64-bit number represented in the IEEE format. It consists of an optionally signed integer portion, a decimal point, a fraction portion, and an exponent. The precision (single or double) of the constant ultimately depends upon the context in which the constant is assembled.

The following conventions help the assembler disambiguate certain floating-point constants from identifiers beginning with `.` and a digit. A leading `0f` or `0F` identifies a single-precision constant while a leading `0d` or `0D` identifies a double-precision constant. Floating-point constants may begin with one of these prefixes (making the integer portion optional), or they must possess an integer portion.

The fraction portion may be omitted. Either the decimal point and the fraction portion or the exponent may be omitted, but not both. The exponent consists of `e` or `E` followed by an optionally signed integer. For example,

```
-4.3, 25.4367e-10, 0f.15
```

Character Constants

A *single-character constant* consists of a single quote `'` followed by an ASCII character other than backslash (\). The value of the constant is the ASCII code for the character. Special meanings of characters are overridden when used in character constants. For example, `'#` and `';` represent the constants # and `;` , respectively, and do not represent a terminating `'` followed by a comment.

A *special character constant* consists of `'\` followed by another character. The special character constants are listed below.

Constant	Value	Meaning
`'\b`	0x08	backspace
`'\t`	0x09	horizontal tab
`'\n`	0x0a	newline (line feed)
`'\f`	0x0c	form feed
`'\r`	0x0d	carriage return
`'\?`	0x3f	question mark
`'\"`	0x22	double quote
`'\'`	0x27	single quote
`'\\`	0x5c	backslash
`'\`*nnn*	0*nnn*	octal character *nnn*

For example, `'q, '\n, '015`

Expressions

Expressions represent 32-bit, two's complement values. They are built up from symbols, constants, operators, and parentheses. Expressions have types, which are discussed later in this section.

Expression Operators

The following operators are available.

Class	Operator	Function	Comment
binary	`+`	addition	
	`-`	subtraction	
	`*`	multiplication	
	`/`	division	The integer quotient is returned, with truncation performed on the real value
	`&`	bitwise AND	
	`|`	bitwise OR	
	`^`	bitwise XOR	
	`~`	bitwise NOR	(a~b) is equivalent to (a OR (NOT b))
	`<`	logical left shift	(a<b) is a shifted left b bits
	`>`	arithmetic right shift	(a>b) is a shifted right b bits
unary	`-`	negation	
	`~`	one's complement	

Operator Precedence

The precedences of the operators appear next.

	`()`	highest
unary	`~, +, -`	|
	`*, /, <, >`	|
	`|, ^, &`	|
binary	`+, -`	|
		|
		lowest

Binary operators of the same precedence are left-to-right associative. Parentheses may be used to override the default precedences and/or associativity.

Expression Types

The type of an expression depends upon the types of the operators and the operands. The possible expression and identifier types are:

manifest	The value can be computed by the assembler at the time of its appearance.
absolute	The value can be computed by the assembler, though not necessarily at the time of its appearance.
relocatable	The value is relative to the start of a particular section. The memory location represented by the expression is not known at assembly time, but the relative values of two such expressions are known if they refer to the same section.
undefined external	No value is assigned to the expression. It is expected that the values will be determined at link time. The relative values of undefined externals are not known at assembly time.

A manifest value is also an absolute value. All absolute values are also manifest values, except for the difference between two relocatable values.

The following rules determine the type of an expression based upon the types of the operands.

- If both operands are of manifest type, the expression is manifest.
- If both operands are of absolute type, the expression is absolute.
- If one operand is an undefined external, the expression is an undefined external.
- If one operand is absolute, and the other operand is relocatable, the expression is relocatable.
- The difference of two relocatable operands is of absolute type.
- It is not possible for one operand to be manifest while the other is absolute or relocatable.

Expression Values

An *absolute symbol* is defined from a constant, and its value is not affected by the link editor.

Text, *data*, and *bss symbols* have values which indicate their displacements from the beginning of the **`.text`**, **`.data`**, or **`.bss`** sections, respectively. Text, data, and bss symbols may change in value if the assembler output is link-edited.

At the beginning of assembly, the value of the location counter **`.`** is the beginning displacement of the **`.text`** section. After the first **`data`** directive is seen, the value of **`.`** becomes the beginning displacement of the **`.data`** section.

Symbols which are declared **`global`** have global visibility. Such a symbol may be defined in the current assembly, or it may be defined externally to the current assembly. If it is defined in the current assembly as an absolute, a text, a data, or a bss symbol, the symbol may be used as if it were not globally visible. Its value and type may be used by the link editor to satisfy external references to the symbol. If the symbol is not defined in the current assembly, the link editor will regard it as an external reference to a global definition of the symbol outside the current assembly.

Assembler Directives

Assembler directives (pseudo-ops) specify location counter control, section switching, data initialization, symbol definitions, symbolic debugging information, and miscellaneous operations. The following notation is used:

`{`*directive* `|` *.directive*`}` `[`*operand*`]...`

directive and *.directive* are acceptable assembly mnemonics, and *operand* is the kind of operand accepted by the directive.

Location Counter Control

`{align | .align}` *alignment*

The location counter is adjusted so that its value, modulo the specified *alignment*, is zero. Bytes between the current location counter and the new (aligned) value are filled with zeroes (`\0`). *alignment* is the base-2 logarithm of the desired alignment. *alignment* is of manifest type. For example,:

```
align 3    /* align the location counter to an 8-byte
    boundary */
```

`.org` *counter*

The location counter is set to *counter*, which must be defined and must not exceed the current value of the location counter. Its recommended use is to set the location counter at a known offset beyond an already-seen label. The directive should be in the same section as the referenced label. A constant *counter* may be used, but the assembler will produce a warning message. For example,

```
label: .long 5; .org label+30 /* change the location
    counter to 30 past the label */
```

`{zero | .space}` *number*

number bytes of zeroes (`\0`) are assembled at the current location counter. *number* must be non-negative. It is of manifest type. For example,

```
zero 24   /* assemble 24 bytes of zeroes */
```

Section Switching

{**`text | .text`**}

The location counter is changed to the next available value in the **`.text`** section. Before the first section directive is encountered in an input file, assembly is by default directed into the **`.text`** section.

{**`data | .data`**}

The location counter is changed to the next available value in the **`.data`** section.

`section` *identifier*`[,`*attributes*`][,`*sectiontype*`]`

Succeeding bytes are assembled into the section named *identifier*. One or more flags comprise a quoted character string of *attributes* for the section. The *attributes* flags are optional. The attributes are indicated in the `sh_flags` entry of the section header. The assembler permits another optional parameter, *sectiontype*, which is indicated in the `sh_type` entry in the section header. This section is created, if it does not already exist, with the given *attributes* and *sectiontype*. If the same section is specified by more than one **`section`** directive, the last value of *attributes* and *sectiontype* is assigned to the section.

Any combination of the following flags can be specified in the *attributes* string.

`w`	Set the `SHF_WRITE` flag (0x1)
`x`	Set the `SHF_ALLOC` flag (0x2)
`a`	Set the `SHF_EXECINSTR` flag (0x4)

The assembler permits one of the following flags to be specified as *sectiontype*. The assembler requires that the given value be preceded with an `@`.

`progbits`	The section may contain data
`nobits`	The section contains no data
`symtab`	The section is a symbol table
`strtab`	The section is a string table
`note`	The section is a comment section
`vendor`	The section is a Concurrent Computer Corporation vendor section

A hexadecimal integer constant may also be specified as *sectiontype*, provided it is preceded by `#` or `@`, as described above.

Some of the flags do not have meaning in a PowerMAX OS environment. They are provided for compatibility with other systems.

As an example,

```
section mysect,"a",@progbits    /* specify ’mysect’ as
    SHF_ALLOC and SHT_PROGBITS */
```

`previous`

This directive exchanges the current section and the previous section.

At any point in the assembly, both a *current* section and a *previous* section are in effect. Initially, the current section is *text* and the previous section is undefined. A **`text`**, **`data`**, or **`section`** operation causes the current section to become the previous section and the operation-specified section to become the current section.

Data Initialization

{**`byte | .byte`**} *value*`[,`*value*`]...`

The specified *value*(s) are assembled into consecutive 1-byte locations. Each *value* is of manifest type and is in the range -(2^7) to 2^8-1. For example,

```
byte 21, -43
```

`ubyte` *expression*`[,`*expression*`]...`

The specified *expression*(s) are assembled into consecutive 1-byte locations. Each *expression* is of absolute or relocatable type or is an undefined external. Each *expression* is in the range 0 to 2^8-1. For example,

```
ubyte 55, 0
```

`sbyte` *expression*`[,`*expression*`]...`

The specified *expression*(s) are assembled into consecutive 1-byte locations. Each *expression* is of absolute or relocatable type or is an undefined external. Each *expression* is in the range -(2^7) to 2^7-1. For example,

```
sbyte -63,34
```

{**`vbyte | .vbyte`**} *number, expression*

The specified *expression* is assembled into consecutive *number*-byte locations. *expression* is of manifest or absolute type. *number* is in the range 1-4, inclusive. If *expression* requires more than *number* bytes, the left-most bytes are not assembled. For example,

```
.vbyte 3,726
```

{**`half | .word | short | .short`**} *value*`[,`*value*`]...`

The assembler requires that the location counter be evenly divisible by 2 when this directive is used. The specified *value*(s) are assembled into consecutive 2-byte locations. Each *value* is of manifest type and is in the range -(2^{15}) to 2^{16}-1. For example,

```
half 0x56b
```

`uhalf` *expression*`[,`*expression*`]...`

The location counter must be evenly divisible by 2 when this directive is used. The specified *expression*(s) are assembled into consecutive 2-byte locations. Each *expression* is of absolute or relocatable type or is an undefined external. Each *expression* is in the range 0 to 2^{16}-1. For example,

```
uhalf 1078,457,3
```

`shalf` *expression*`[,`*expression*`]...`

The location counter must be evenly divisible by 2 when this directive is used. The specified *expression*(s) are assembled into consecutive 1-byte locations. Each *expression* is of absolute or relocatable type or is an undefined external. Each *expression* is in the range -(2^{15}) to 2^{15}-1. For example,

```
shalf -20345,26
```

`uahalf` *value*`[,`*value*`]...`

There is no restriction on the divisibility of the location counter when this directive is used. The specified *value*(s) are assembled into consecutive 2-byte locations. Each *value* is of absolute type and is in the range -(2^{15}) to 0 to 2^{16}-1. For example,

```
uahalf 7823,-40201
```

{**`word | .int | .long`**} *value*`[,`*value*`]...`

The location counter must be evenly divisible by 4 when this directive is used. The specified *value*(s) are assembled into consecutive 4-byte locations. Each *value* is of manifest type and is in the range -(2^{31}) to 2^{32}-1. For example,

```
word -3, 759323, 0
```

`uaword` *expression*`[,`*expression*`]...`

There is no restriction on the divisibility of the location counter when this directive is used. The specified *expression*(s) are assembled into consecutive 4-byte locations. Each *expression* is of absolute or relocatable type or is an undefined external. Each *expression* is in the range -(2^{31}) to 2^{32}-1. For example,

```
uaword 1078,457,-108324
```

{**`float | .float`**} *floatconst*`[,`*floatconst*`]...`

The location counter must be evenly divisible by 4 when this directive is used. The specified *floatconst*(s) are assembled into consecutive 4-byte locations. Each *floatconst* is in the range of IEEE single-precision numbers. For example,

```
float 3.1415, 0.0
```

{`double | .double`} *floatconst*`[,`*floatconst*`]...`

The location counter must be evenly divisible by 8 when this directive is used. The specified *floatconst*(s) are assembled into consecutive 8-byte locations. Each *floatconst* is in the range of IEEE double-precision numbers. For example,

```
double -1.5, 2.34e31
```

{`string | .ascii`} *string*`[,`*string*`]...`

The specified *string*(s) are assembled into consecutive locations--one character of the string per byte. The quoted string is regarded as a C-style string. The leading and the terminating double quotes are not assembled, and the string is not appended with a trailing null byte (`\0`). For example,

```
string "several bytes"
```

`.asciiz` *string*`[,`*string*`]...`

The specified *string*(s) are assembled into consecutive locations--one character of the string per byte. The quoted string is regarded as a C-style string. The leading and the terminating double quotes are not assembled, and the string is appended with a trailing null byte (`\0`). For example,

```
asciiz "error in format\n", "syntax error\n"
```

Symbol Definitions

{`def | .def | set | .set`} *identifier*`,`*expression*

The assembler requires that *expression* be of absolute or relocatable type. A new symbol, *identifier*, is created, and its value is set to the value of *expression*. For example,

```
def temp,2*4
    /* create a variable 'temp', giving it the value 8 */
```

{`global | .globl`} *identifier*

identifier is made externally visible. If *identifier* is defined in this assembly, its definition may be used by the link editor to resolve external references to it. If *identifier* is not defined in this assembly, the link editor must locate an external definition to satisfy its external reference. For example,

```
global sub    /* give 'sub' external visibility */
```

{`extern | .extern`} *identifier*

identifier is regarded as being defined in another source file. For example,

```
extern var
 /* identify 'var' as defined in another source file. */
```

{**comm | .comm**} *identifier*, *size*[, *alignment*]

identifier is made externally visible and is to be assigned to a common area of *size* bytes. If *identifier* is not defined at link time, the link editor assigns it to the **.bss** section. *identifier* becomes relocatable. *size* is of manifest type. The optional third argument, *alignment*, is of manifest type and must be a power of two. It has the meaning as *alignment* in the **align** directive, above. If *alignment* is not specified, the alignment of *identifier* is 1 or 2 when *size* is 1 or 2, respectively; otherwise, the alignment is to an 8-byte boundary. For example,

```
comm block,20    /* define a common area ’block’ of
                  size 20 bytes, on an 8-byte boundary */
```

{**bss | .bss**} *identifier*, *size*[, *alignment*]

identifier is made externally invisible but internally visible. It is *size* bytes long and is assigned to the **.bss** section. *alignment* is optional and must be a power of two, if present. If *alignment* is missing the alignment is regarded as 1-byte. Both *size* and *alignment* are of manifest type. For example,

```
bss var,10,4    /* define a .bss variable ’var’, size
                     10 bytes, on a 4-byte boundary */
```

local *identifier*[, *identifier*]...

Each *identifier* is defined in the input file and not accessible to other files. Any default binding for *identifier* is overridden by this directive. For example,

```
local local_var    /* declare a local variable
                    ’local_var’ */
```

weak *identifier*[, *identifier*]...

Each *identifier* is declared to be a weak global identifier. It is either defined externally or defined in the input file and accessible in other files. Any default binding for *identifier* is overridden by this directive. For example,

```
weak _sub    /* give ’_sub’ weak binding */
```

NOTE

The assembler permits the use of at most one of **global**, **local**, and **weak** for each symbol in the input file.

ELF Symbol Attributes

These directives provide attributes for symbols. Refer to Chapter 22 (“Executable and Linking Format (ELF)”) for information about the symbol table.

`type` *identifier*`,`*type*

identifier is declared with type *type*. The assembler permits one of the following flags to be specified as *type*. The assembler requires that the given value be preceded with a `@`.

`no_type`	no specified type
`object`	a data object
`function`	a function or other executable code

For example,

```
type abc,@object    /* associate 'abc' with a data
                         object */
```

`size` *identifier*`,`*size*

The size *size* is associated with *identifier*. *size* specifies the size in bytes and is of absolute type. For example,

```
size 6    /* indicate the identifier has size 6 */
```

`version` *value*

The quoted string *value* is compared with an internal assembler version string. If *value* is lexicographically greater than the internal string, the assembler produces a fatal error message and exits.

This directive is optional. If present, it must appear first in the assembly file. The only acceptable *values* are **`"03.00"`** and **`"02.00"`**. **`"02.00"`** suppresses the automatic removal of a leading underscore from labels and alphanumeric labels. Additionally, **`"03.00"`** requires that # be prefixed to certain keywords, as described throughout this chapter.

Miscellaneous Operations

{**`file`** `|` **`.file`**} *file*

The quoted string *file* is placed in the object file's symbol table. The leading and the terminating double quotes are not assembled, and the string is not appended with a trailing null byte (`\0`). *file* is of length 1-255 characters, inclusive. If the assembler **`-A`** option is used, however, *file* may be of length 1-800 characters, inclusive. Only one **`file`** directive may be specified in an assembly file. If the **`-A`** option is used, however, multiple **`file`** directives may be specified. For example,

```
file "source.c"    /* place the file name 'source.c' in
                        the symbol table */
```

`ident` *string*

string is assembled into the **`.comment`** section. It is regarded as a C-style string. The leading and the terminating double quotes are not assembled, and the string is appended with a trailing null byte (`\0`). This directive is typically used to provide revision level tracking information. For example,

```
ident "revision 5.1.3"    /* place the string in the
                             .comment section */
```

`fp_spec_exec`

This directive indicates that the assembly code contains floating-point instructions that are executed in a speculative manner. (See the discussion of speculative execution in Chapter 20.) Modules that speculatively execute floating-point instructions could erroneously raise floating-point exceptions, making it necessary to link programs with all floating-point exceptions disabled. (See the discussion of the **`-Qfpexcept=`** option in Chapter 3.)

```
fp_spec_exec    /* indicate that floating-point
                   instructions are speculatively
                   executed.*/
```

Summary of Directives Mnemonics

Table 2-1 summarizes the available directives.

Table 2-1. Available Directives

Mnemonic(s)	Argument(s)
`align, .align`	*alignment*
`.org`	*counter*
`zero, .space`	*number*
`text, .text`	
`data, .data`	
`section`	*identifier*[,*attributes*][,*sectiontype*]
`previous`	
`byte, .byte`	*value*[,*value*]...
`ubyte`	*expression*[,*expression*]...
`sbyte`	*expression*[,*expression*]...
`vbyte, .vbyte`	*number, expression*
`half, .word`	*value*[,*value*]...
`short, .short`	*value*[,*value*]...
`uhalf`	*expression[,expression]*...
`shalf`	*expression*[,*expression*]...
`uahalf`	*value*[,*value*]...
`word, .int, .long`	*value*,[*value*]...

Table 2-1. Available Directives (Cont.)

Mnemonic(s)	Argument(s)
uaword	*expression*[,*expression*]...
float, .float	*floatconst*[,*floatconst*]...
double, .double	*floatconst*[,*floatconst*]...
string, .ascii	*string*[,*string*]...
.asciiz	*string*[,*string*]...
def, .set	*identifier*,*expression*
.def, set	*identifier*,*expression*
global, .globl	*identifier*
extern, .extern	*identifier*
comm, .comm	*identifier*,*size*[,*alignment*]
bss, .bss	*identifier*,*size*[,*alignment*]
local	*identifier*[,*identifier*]...
weak	*identifier*[,*identifier*]...
type	*identifier*,*type*
size	*identifier*,*size*
version	*value*
file, .file	*file*
ident	*string*
fp_spec_exec	

Example

The following C function could be assembled to the assembly source code shown below. Assembly source that is accepted by the assembler is used in this example.

```
sub(i) {
    if (i > 0) {
        printf (" the value of i = %d \n", i);
   }
}

        version "02.00"
        file    "example.c"
        data
        align   3
```

```
lit_lab:
        string  "the value of i = %d\n\000"
        text
        align   2
        global  sub

sub:
        type    sub,@function
        size    sub,..sub_sub_end - sub
        addi    r1,r1,-80
        mflr    r13        # return address
        stw     r13,88(r1)   ; mr   r4,r3

..sub_sub_:
        # line 3
        cmpwi   crf1,r4,0
        ble     crf1,@L6

# line 4
        lis     r3,uhi16(lit_lab)
        ori     r3,r3,lo16(lit_lab)
        bl      printf

@L6:
        lwz     r13,88(r1)
        mtlr    r13
        addi    r1,r1,80
        blr

..sub_sub_end:
@L12:
        section .tdesc,"x"
        word    0x42
        word    0x1
        word    ..sub_sub_
        word    @L12
        word    0x10000021
        word    0x50,0x8,0xfffffff0
```

Position-Independent Code

Assembly Syntax

The assembly language is extended to support position-independent code, which is used in dynamic linking. (See Chapter 4 ("Link Editor and Linking") for information on dynamic linking and shared object files.) The following expressions are explained, and some of these extensions are used in the example that follows. The assembler requires that `@` be used in these expressions (e.g., *s*`@`**`got`**). The `@` is used in the explanations that follow.

s@**got**	The address of a global offset table entry for symbol *s*.
p@**gotp**	The address of a global offset table procedure entry for the procedure named by the symbol *p*.
p@**plt**	An address to which control can be transferred to invoke the procedure named by symbol *p*. It is either the address of *p* or the address of a procedure linkage table entry for *p*.
s@**rel**	The difference between the value of the symbol *s* and the addressing base for the object containing the expression. The value of the symbol *s* must represent an address in the object containing the expression.
s@**got_rel**	The difference between the address denoted by *s*@**got** and the addressing base for the object containing the expression.
p@**gotp_rel**	The difference between the address denoted by *p*@**gotp** and the addressing base for the object containing the expression.
p@**plt_rel**	The difference between the address denoted by *p*@**plt** and the addressing base for the object containing the expression.
s@**abdiff**	The difference between the addressing base for the shared object containing the expression and the value of the symbol *s*. The value of the symbol *s* must represent an address in the object containing the expression.

The *addressing base* refers to a particular virtual address associated with the memory image of a shared object. A position-independent function establishes the addressing base by computing its value and preserving it in a register for use throughout the activation of the function.

Example

The following C code serves to illustrate the difference between position-independent and position-dependent code, at the assembly language level. Assembly source that is accepted by the assembler is used in this example.

```
int global;
int *global_ptr;
sub () {
    extern char * malloc();
    global_ptr = (int *) malloc (20);
    *global_ptr = global;
}
```

Position-Independent			Position-Dependent		
	global	sub		global	sub
sub:			sub:		
	addi	r1,r1,-80		addi	r1,r1,-80
	stw	r16,64(r1)			
	mflr	r13		mflr	r13
	stw	r13,88(r1)		stw	r13,88(r1)
	local	base			
	bl	base			
base:					
	ori	r16,r16,lo16(base@abdiff)			
	mflr	r13			
	add	r16,r16,r3			
	local	be			
be:			be:		
	lis	r3,uhi16(malloc@gotp_rel)			
	ori	r3,r3,lo16(malloc@gotp_rel)			
	lwzx	r4,r16,r3			
	li	r3,lo16(0x14)		li	r3,lo16(0x14)
	mtctr	r4			
	btcrl			bl	malloc
	lis	r4,uhi16(global_ptr@got_rel)		lis	r4,hi16(global_ptr)
	ori	r4,r4,lo16(global_ptr@got_rel)			
	lwzx	r4,r16,r4			
	stw	r3,0(r4)		stw	r3,lo16(global_ptr)(r4)
	lis	r3,uhi16(global@got_rel)		lis	r4,hi16(global)
	ori	r3,r3,lo16(global@got_rel)		lwz	r4,lo16(gloabl)(r4)
	lwzx	r3,r16,r3			
	lwz	r4,0(r3)			
	lis	r3,uhi16(global_ptr@got_rel)			
	ori	r3,r3,lo16(global_ptr@got_rel)			
	lwzx	r3,r16,r3			
	lwz	r3,0(r3)			
	stw	r4,0(r3)		stw	r4,0(r3)
	lwz	r16,64(r1)			
	lwz	r13,88(r1)		lwz	r13,88(r1)
	mtlr	r13		mtlr	r13
	addi	r1,r1,80		addi	r1,r1,80
	blr			blr	
en:			en:		

Position-Independent		Position-Dependent	
section	.tdesc,"x"	section	.tdesc,"x"
word	0x42	word	0x42
word	0x2	word	0x1
word	be@rel	word	be
word	en@rel	word	en
word	0x1020021	word	0x1000021
word	0x50	word	0x50
word	0x8	word	0x8
word	0xfffffff0	word	0xfffffff0

For executable code in a shared object to be shared among multiple processes using that shared object, it must reference symbols and data in a position-independent manner. In the code above, the addressing base is computed into register `r14`.

Because each process will have its own, private copy of the global offset table for procedures, the address of `global_ptr`, specific to a process, can be obtained from the process' private copy of the table. The `global_ptr`**`@got_rel`** syntax directs the assembler to produce relocation information that the link editor will use. The link editor will establish an offset, in the global offset table, which will contain the address of `global_ptr`. The value in register `r3` contains a byte offset from the addressing base to this location in the table. The value in `r5` is the address of `global_ptr`. Thus, an extra level of indirection is needed to obtain the address of the variable. An explanation of the use of **`@gotp_rel`**, for referencing `malloc`, is similar.

The **`be@rel`** syntax directs the assembler and the link editor to produce a value which is the difference between the address of the symbol **`be`** and the addressing base of the shared object. A consumer of this information, such as a debugger, would need to dynamically add the addressing base to this difference to determine the actual address of the symbol.

The local directives are needed to indicate that the symbols are to be regarded as inaccessible from other files and shared objects.

3
PowerPC Instruction Set Summary

3 PowerPC Instruction Set Summary

This chapter summarizes the instruction sets of the PowerPC 601, 602, 603, 603e, 604, 604e, 620, and 75021 microprocessors. Instructions specific to or excluded from other members of the PowerPC family are not documented here. These processors are documented to assist porting between PowerPC implementations. These tables are based on preliminary documentation from the chip manufacturers. The information contained is subject to change without notice. The following special notation conventions apply to tables in this chapter only.

In the PowerPC Mnemonic column:

Bold mnemonics	Signify instructions defined for 64-bit implementations only.
Italic mnemonics	Signify extended mnemonics added by Concurrent Computer Corporation that are not present in IBM or Motorola documentation.
Small mnemonics	Signify IBM RS/6000 POWER™ instructions that are provided on the PowerPC 601 systems for compatibility purposes.

In all columns except the Syntax of Operands column, the following codes that represent variations of the instructions appear:

[o]	Cause the SO and OV bits to be set in the fixed-point exception register.
[.]	For integer instructions, cause crf0 to be set as though the result were compared to zero; for floating-point instructions, cause crf1 to be set with the high order four bits of the floating-point status and control register.
[l]	Cause link register to be set to the return address.
[a]	Cause the displacement to be taken as an absolute address.
[s]	Cause the floating-point result to be rounded to single-precision.
[u]	Cause rA to be updated with the effective address of the load or store.

In the Syntax of Operands column:

[*operand*]	Signify an operand the assembler allows you to omit. This feature is not documented in the IBM documentation, but is a carry over from the Rios assembly language.

In the Description column:

operand defaults to *value*	Signifies the default value for an omitted operand.
(optional)	Signifies instructions defined as optional in the PowerPC architecture definition.
(optional)**(not on xxx)**	Signifies implementations that do not include the optional instruction.
(xxx only)	Signifies instructions that are specific to a particular implementation but are not part of the PowerPC architecture definition.
xxx[e]	Signifies both xxx and xxxe.

In the RS/6000 POWER Mnemonic column:

"	Means the RS/6000 POWER mnemonic is spelled the same as the PowerPC mnemonic.

PowerPC Instruction Set

Table 3-1. PowerPC Instruction Set

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
abs[o][.]	rT,rA	Absolute Value **(601 only)**	"
add[o][.]	rT,rA,rB	Add	cax[o][.]
addc[o][.]	rT,rA,rB	Add Carrying	a[o][.]
adde[o][.]	rT,rA,rB	Add Extended	ae[o][.]
addi	rT,*rA*,SI	Add Immediate	cal
addic[.]	rT,rA,SI	Add Immediate Carrying	ai[.]
addis	rT,*rA*,SI	Add Immediate Shifted	cau
addme[o][.]	rT,rA	Add to Minus One Extended	ame[o][.]
addze[o][.]	rT,rA	Add to Zero Extended	aze[o][.]
and[.]	rA,rS,rB	AND	"
andc[.]	rA,rS,rB	AND with Complement	"
andi.	rA,rS,UI	AND Immediate	andil.

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
andis.	rA,rS,UI	AND Immediate Shifted	andiu.
b[l][a]	LI	Branch	"
bc[l][a]*	BO,BI,BD	Branch Conditional	"
bcctr[l]*	BO,BI	Branch Conditional to Count Register	bcc[l]
bclr[l]*	BO,BI	Branch Conditional to Link Register	bcr[l]
bctr[l]	-	Branch to Count Register Same as: bcctr[l] 20,0	"
bdnz[l][a]*	BD	Branch Decrement Count Non-Zero Same as: bc[l][a] 16,0,BD	bdn[l][a]
*bdnzCC[l][a]**	[crfA,]BD	Branch Decrement Count Non-Zero on CC crfA defaults to crf0 Same as: bc[l][a] BO,BI,BD	bdnCC
*bdnzCClr[l]**	[crfA]	Branch Decrement Count Non-Zero on CC to LR crfA defaults to crf0 Same as: bclr[l] BO,BI,BD	
bdnzf[l][a]*	BI,BD	Branch Decrement Count Non-Zero False Same as: bc[l][a] 0,BI,BD	
bdnzflr[l]*	BI	Branch Decrement Count Non-Zero False to LR Same as: bclr[l] 0,BI	
bdnzlr[l]*	-	Branch Decrement Count Non-Zero to LR Same as: bclr[l] 16,0	bdnr[l]
bdnzt[l][a]*	BI,BD	Branch Decrement Count Non-Zero True Same as: bc[l][a] 8,BI,BD	
bdnztlr[l]*	BI	Branch Decrement Count Non-Zero True to LR Same as: bclr[l] 8,BI	
bdz[l][a]*	BD	Branch Decrement Count Zero Same as: bc[l][a] 18,0,BD	"
*bdzCC[l][a]**	[crfA,]BD	Branch Decrement Count Zero on Condition Code crfA defaults to crf0 Same as: bc[l][a] BO,BI,BD	bdzCC
*bdzCClr[l]**	[crfA]	Branch Decrement Count Non-Zero on CC to LR crfA defaults to crf0 Same as: bclr[l] BO,BI,BD	
bdzf[l][a]*	BI,BD	Branch Decrement Count Zero False Same as: bc[l][a] 2,BI,BD	
bdzflr[l]*	BI	Branch Decrement Count Zero False to LR Same as: bclr[l] 2,BI	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
bdzlr[l]*	-	Branch Decrement Count Zero to LR Same as: bclr[l] 18,0	bdzr[l]
bdzt[l][a]*	BI,BD	Branch Decrement Count Zero True Same as: bc[l][a] 10,BI,BD	
bdztlr[l]*	BI	Branch Decrement Count Zero True to LR Same as: bclr[l] 10,BI	
bf[l][a]*	BI,BD	Branch False Same as: bc[l][a] 4,BI,BD	bbf[l][a]
bfctr[l]*	BI	Branch False to Count Register Same as: bcctr[l] 4,BI	bbfc[l]
bflr[l]*	BI	Branch False to Link Register Same as: bclr[l] 4,BI	bbfr[l]
blr[l]	-	Branch to Link Register Same as: bclr[l] 20,0	br[l]
bt[l][a]*	BI,BD	Branch True Same as: bc[l][a] 12,BI,BD	bbt[l][a]
btctr[l]*	BI	Branch True to Count Register Same as: bcctr[l] 12,BI	bbtc[l]
btlr[l]*	BI	Branch True to Link Register Same as: bclr[l] 12,BI	bbtr[l]
bCC[l][a]*	[crfA,]BD	Branch on Condition Code crfA defaults to crf0 Same as: bc[l][a] BO,BI,BD	"
bCCctr[l]*	[crfA]	Branch on Condition Code to Count Register crfA defaults to crf0 Same as: bcctr[l] BO,BI	bCCc[l]
bCClr[l]*	[crfA]	Branch on Condition Code to Link Register crfA defaults to crf0 Same as: bclr[l] BO,BI	bCCr[l]
clcs	rT,rA	Cache Line Compute Size **(601 only)**	"
clrldi[.]	rA,rS,n	Clear Left Doubleword Immediate Same as: rldicl[.] rA,rS,0,n	
clrlsdi[.]	rA,rS,b,n	Clear Left and Shift Doubleword Immediate Same as rldicr[.] rA,rS,n,b-n	
clrlslwi[.]	rA,rS,b,n	Clear Left and Shift Left Word Immediate $n \le b \le 31$ Same as: rlwinm[.] rA,rS,n,b-n,31-n	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
clrlwi[.]	rA,rS,n	Clear Left Word Immediate *n* < 31 Same as: rlwinm[.] rA,rS,0,n,31	
clrrdi[.]	rA,rS,n	Clear Right Doubleword Immediate Same as: rldicl[.] rA,rS,0,63-n	
clrrwi[.]	rA,rS,n	Clear Right Word Immediate *n* < 31 Same as: rlwinm[.] rA,rS,0,0,31-n	
cmp	[crfT,]L,rA,rB	Compare crfT defaults to crf0	
cmpd	[crfT,]rA,rB	Compare Doubleword crfT defaults to crf0 Same as: cmp crfT,1,rA,rB	
cmpdi	[crfT,]rA,SI	Compare Doubleword Immediate crfT defaults to crf0 Same as: cmpi crfT,1,rA,SI	
cmpi	[crfT,]L,rA,SI	Compare Immediate crfT defaults to crf0	
cmpw	[crfT,]rA,rB	Compare Word crfT defaults to crf0 Same as: cmp crfT,0,rA,rB	cmp
cmpwi	[crfT,]rA,SI	Compare Word Immediate crfT defaults to crf0 Same as: cmpi crfT,0,rA,SI	cmpi
cmpl	[crfT,]L,rA,rB	Compare Logical crfT defaults to crf0	
cmpld	[crfT,]rA,rB	Compare Logical Doubleword crfT defaults to ctf0 Same as: cmpl crfT,1,rA,rB	
cmpldi	[crfT,]rA,UI	Compare Logical Doubleword Immediate crfT defaults to crf0 Same as: cmpli crfT,1,rA,UI	
cmpli	[crfT,]L,rA,UI	Compare Logical Immediate crfT defaults to crf0	
cmplw	[crfT,]rA,rB	Compare Logical Word crfT defaults to crf0 Same as: cmpl crfT,0,rA,rB	cmpl
cmplwi	[crfT,]rA,UI	Compare Logical Word Immediate crfT defaults to crf0 Same as: cmpli crfT,0,rA,UI	cmpli

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
cntlzd[.]	rA,rS	Count Leading Zeros Doubleword	
cntlzw[.]	rA,rS	Count Leading Zeros Word	cntlz[.]
crand	BT,BA,BB	Conditional Register AND	"
crandc	BT,BA,BB	Conditional Register AND with Complement	"
crclr	BT	Conditional Register Clear Bit Same as: crxor BT,BT,BT	
creqv	BT,BA,BB	Conditional Register Equivalent	"
crmove	BT,BA	Conditional Register Move Same as: cror BT,BA,BA	
crnand	BT,BA,BB	Conditional Register NOT AND	"
crnor	BT,BA,BB	Conditional Register NOT OR	"
crnot	BT,BA	Conditional Register NOT Same as: crnor BT,BA,BA	
cror	BT,BA,BB	Conditional Register OR	"
crorc	BT,BA,BB	Conditional Register OR with Complement	"
crset	BT	Conditional Register Set Bit Same as: creqv BT,BT,BT	
crxor	BT,BA,BB	Conditional Register Exclusive OR	"
dcbf	*rA*,rB	Data Cache Block Flush	
dcbi	*rA*,rB	Data Cache Block Invalidate Supervisor Level	
dcbst	*rA*,rB	Data Cache Block Store	
dcbt	*rA*,rB	Data Cache Block Touch	
dcbtst	*rA*,rB	Data Cache Block Touch for Store	
dcbz	*rA*,rB	Data Cache Block set to Zero	dclz
div[o][.]	rT,rA,rB	Divide **(601 only)**	
divd[o][.]	rT,rA,rB	Divide Doubleword	
divdu[o][.]	rT,rA,rB	Divide Doubleword Unsigned	
divs[o][.]	rT,rA,rB	Divide Short **(601 only)**	
divw[o][.]	rT,rA,rB	Divide Word	
divwu[o][.]	rT,rA,rB	Divide Word Unsigned	
doz[o][.]	rT,rA,rB	Difference or Zero **(601 only)**	
dozi	rT,rA,SI	Difference or Zero Immediate **(601 only)**	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
dsa	-	Disable Supervisor Access **(602 only)**	
eciwx	rT,*rA*,rB	External Control In Word Indexed (optional) **(not on 602)**	
ecowx	rS,*rA*,rB	External Control Out Word Indexed (optional) **(not on 602)**	
eieio	-	Enforce In-order Execution of I/O	
eqv[.]	rA,rS,rB	Equivalent	
esa	-	Enable Supervisor Access **(602 only)**	
extldi[.]	rA,rS,n,b	Extract and Left Justify Doubleword Immediate Same as: rldicr[.] rA,rS,b,n-1	
extlwi[.]	rA,rS,n,b	Extract and Left Justify Word Immediate Same as: rlwinm[.] rA,rS,b,0,n-1	
extrdi[.]	rA,rS,n,b	Extract and Right Justify Doubleword Immediate Same as: rldicl[.] rA,rS,b+n,64-n	
extrwi[.]	rA,rS,n,b	Extract and Right Justify Word Immediate Same as: rlwinm[.] rA,rS,b+n,32-n,31	
extsb[.]	rA,rS	Extend Sign Byte	
extsh[.]	rA,rS	Extend Sign Halfword	exts[.]
extsw[.]	rA,rS	Extend Sign Word	
fabs[.]	fT,fB	Floating Absolute Value	"
fadd[s][.]	fT,fA,fB	Floating Add **(double precision not on 602)**	fa[.]
fcfid[.]	fT,fB	Floating Convert From Integer Doubleword	
fcmpo	[crfT,]fA,fB	Floating Compare Ordered crfT defaults to crf0	"
fcmpu	[crfT,]fA,fB	Floating Compare Unordered crfT defaults to crf0	"
fctid[.]	fT,fB	Floating Convert to Integer Doubleword	
fctidz[.]	fT,fB	Floating Convert to Integer Doubleword with round toward Zero	
fctiw[.]	fT,fB	Floating Convert to Integer Word	
fctiwz[.]	fT,fB	Floating Convert to Integer Word with round toward Zero	
fdiv[s][.]	fT,fA,fB	Floating Divide **(double precision not on 602)**	fd[.]
fmadd[s][.]	fT,fA,fB,fC	Floating Multiply-Add **(double precision not on 602)**	fma[.]
fmr[.]	fT,fB	Floating Move Register	"

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
fmsub[s][.]	fT,fA,fB,fC	Floating Multiply-Subtract **(double precision not on 602)**	fms[.]
fmul[s][.]	fT,fA,fC	Floating Multiply **(double precision not on 602)**	fm[.]
fnabs[.]	fT,fB	Floating Negate Absolute Value	"
fneg[.]	fT,fB	Floating Negate	"
fnmadd[s][.]	fT,fA,fB,fC	Floating Negate Multiply-Add **(double precision not on 602)**	fnma[.]
fnmsub[s][.]	fT,fA,fB,fC	Floating Negate Multiply-Subtract **(double precision not on 602)**	fnms[.]
fres[.]	fT,fB	Floating Reciprocal Estimate Single (optional) **(not on 601)**	
frsp[.]	fT,fB	Floating Round to Single-Precision **(double precision not on 602)**	"
frsqrte[.]	fT,fB	Floating Reciprocal Square Root Estimate (optional) **(not on 601)**	
fsel[.]	fT,fA,fC,fB	Floating Select (optional) **(not on 601)**	
fsqrt[s][.]	fT,fB	Floating Square Root (optional) **(Not on 601, 602, 603[e], 604[e])**	fsqrt[.] (RS/6000 POWER2 only)
fsub[s][.]	fT,fA,fB	Floating Subtract **(double precision not on 602)**	fs[.]
icbi	*rA*,rB	Instruction Cache Block Invalidate	
inslwi[.]	rA,rS,n,b	Insert from Left Word Immediate Same as: rlwimi[.] rA,rS,32-b,b,b+n-1	
insrdi[.]	rA,rS,n,b	Insert from Right Doubleword Immediate Same as: rldimi[.] rA,rS,64-b-n,b	
insrwi[.]	rA,rS,n,b	Insert from Right Word Immediate Same as: rlwimi[.] rA,rS,32-b-n,b,b+n-1	
isync	-	Instruction Synchronize	ics
la	rT,D(*rA*)	Load Address Same as: addi rT,rA,D	
lax	rT,*rA*,rB	Load Address Indexed Same as: add rT,rA,rB	
lbz[u]	rT,D(*rA*)	Load Byte and Zero	"
lbz[u]x	rT,*rA*,rB	Load Byte and Zero Indexed	"
ld[u]	rT,D(*rA*)	Load Doubleword	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
ld[u]x	rT,*rA*,rB	Load Doubleword Indexed	
ldarx	rT,*rA*,rB	Load Doubleword And Reserve Indexed	
lfd[u]	fT,D(*rA*)	Load Floating-Point Double **(not on 602)**	"
lfd[u]x	fT,*rA*,rB	Load Floating-Point Double Indexed **(not on 602)**	"
lfs[u]	fT,D(*rA*)	Load Floating-Point Single	"
lfs[u]x	fT,*rA*,rB	Load Floating-Point Single Indexed	"
lha[u]	rT,D(*rA*)	Load Halfword Algebraic	"
lha[u]x	rT,*rA*,rB	Load Halfword Algebraic Indexed	"
lhbrx	rT,*rA*,rB	Load Halfword Byte-Reverse Indexed	"
lhz[u]	rT,D(*rA*)	Load Halfword and Zero	"
lhz[u]x	rT,*rA*,rB	Load Halfword and Zero Indexed	"
li	rT,SI	Load Immediate Same as: addi rT,0,SI	lil
lis	rT,SI	Load Immediate Shifted Same as: addis rT,0,SI	liu
lmw	rT,D(*rA*)	Load Multiple Word	lm
lscbx[.]	rT,rA,rB	Load String And Compare Byte Indexed **(601 only)**	"
lswi	rT,*rA*,NB	Load String Word Immediate **(not on 602)**	lsi
lswx	rT,*rA*,rB	Load String Word Indexed **(not on 602)**	lsx
lwa	rT,D(*rA*)	Load Word Algebraic	
lwarx	rT,*rA*,rB	Load Word And Reserved Indexed	
lwax	rT,*rA*,rB	Load Word Algebraic Indexed	
lwbrx	rT,*rA*,rB	Load Word Byte-Reverse Indexed	lbrx
lwz[u]	rT,D(*rA*)	Load Word and Zero	l[u]
lwz[u]x	rT,*rA*,rB	Load Word and Zero Indexed	l[u]x
maskg[.]	rA,rS,rB	Mask Generate **(601 only)**	"
maskir[.]	rA,rS,rB	Mask Insert From Register **(601 only)**	"
mcrf	crfT,crfA	Move Condition Register Field	"
mcrfs	crfT,BFA	Move to Condition Register Field from FPSCR	"
mcrxr	crfT	Move to Condition Register Field from XER	"
mfasr	rT	Move From Address Space Register Supervisor Level Same as: mfspr rT,280	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mfbatl	rT,n	Move From Block Address Translation Lower n **(601 only)** Supervisor Level Same as: mfspr rT,529+2n	
mfbatu	rT,n	Move From Block Address Translation Upper n **(601 only)** Supervisor Level Same as: mfspr rT,528+2n	
mfbuscr	rT	Move From Bus Control & Status Register **(620 only)** Supervisor Level Same as: mfspr rT,1016	
mfcr	rT	Move From Condition Register	"
mfctr	rT	Move From Count Register Same as: mfspr rT,9	"
mfdabr	rT	Move From Data Address Breakpoint Register **(601, 604[e], 620 only)** Supervisor Level Same as: mfspr rT,1013	
mfdar	rT	Move From Data Address Register Supervisor Level Same as: mfspr rT,19	
mfdbatl	rT,n	Move From Data Block Address Translation Lower n **(not on 601)** Supervisor Level Same as: mfspr rT,537+2n	
mfdbatu	rT,n	Move From Data Block Address Translation Upper n **(not on 601)** Supervisor Level Same as: mfspr rT,536+2n	
mfdcmp	rT	Move From Data TLB Compare **(602, 603[e] only)** Supervisor Level Same as: mfspr rT,977	
mfdec	rT	Move From Decrementer Supervisor Level Same as: mfspr rT,22	"
mfdmiss	rT	Move From Data TLB Miss Address **(602, 603[e] only)** Supervisor Level Same as: mfspr,976	
mfdsisr	rT	Move From Data Storage Interrupt Status Register Supervisor Level Same as: mfspr rT,18	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mfear	rT	Move From External Access Register (optional) **(not on 602)** Supervisor Level Same as: mfspr rT,282	
mfesasrr	rT	Move From Enable Supervisor Access Save and Restore Register **(602 only)** Supervisor Level Same as: mfspr rT,987	
mffpecr	rT	Move From Floating-Point Exception Cause Register (optional) **(not on 601, 602, 603[e], 604[e])** Supervisor Level Same as: mfspr rT,1022	
mffs[.]	fT	Move From FPSCR	"
mfhash1	rT	Move From Primary Hash Address **(602, 603[e] only)** Supervisor Level Same as: mfspr rT, 978	
mfhash2	rT	Move From Secondary Hash Address **(602, 603[e] only)** Supervisor Level Same as: mfspr rT, 979	
mfhid0	rT	Move From Hardware Implementation Dependent 0 **(601, 602, 603[e], 604[e], 620 only)** Supervisor Level Same as: mfspr rT,1008	
mfhid1	rT	Move From Hardware Implementation Dependent 1 **(601, 602, 603e, 604e only)** Supervisor Level Same as: mfspr rT,1009	
mfiabr	rT	Move From Instruction Address Breakpoint Register **(601, 602, 603[e], 604[e], 620 only)** Supervisor Level Same as: mfspr rT,1010	
mfibatl	rT,n	Move From Instruction Block Address Translation Lower n **(not on 601)** Supervisor Level Same as: mfspr rT,529+2n	
mfibatu	rT,n	Move From Instruction Block Address Translation Upper n **(not on 601)** Supervisor Level Same as: mfspr rT,528+2n	
mfibr	rT	Move From Interrupt Base Register **(602 only)** Supervisor Level Same as: mfspr rT,986	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mficmp	rT	Move From Instruction TLB Compare **(602, 603[e] only)** Supervisor Level Same as: mfspr rT,981	
mfictc	rT	Move From Instruction Cache-Throttling Control Register **(750 only)** Supervisor Level Same as: mfspr rT,1019	
mfimiss	rT	Move From Instruction TLB Miss Address **(602, 603[e] only)** Supervisor Level Same as: mfspr rT,980	
mfl2cr	rT	Move From L2 Control Register **(620 only)** Supervisor Level Same as: mfspr rT,1017	
mfl2sr	rT	Move From L2 Status Register **(620 only)** Supervisor Level Same as: mfspr rT,1018	
mflr	rT	Move From Link Register Same as: mfspr rT,8	"
mflt	rT	Move From Integer Tag Register **(602 only)** Supervisor Level Same as: mfspr rT,1022	
mfmmcr0	rT	Move From Mask Register **(604[e], 620 only)** Supervisor Level Same as: mfspr rT,952	
mfmmcr0rd	rT	Move From Mask Register/Read Only **(620 only)** Supervisor Level Same as: mfspr rT,779	
mfmmcr1	rT	Move From Mask Register 1 **(604e only)** Supervisor Level Same as: mfspr rT,956	
mfmq	rT	Move From Multiply-Quotient Register **(601 only)** Same as: mfspr rT,0	"
mfmsr	rT	Move From Machine State Register Supervisor Level	"
mfpir	rT	Move From Processor ID Register (optional) **(601, 604[e], 620 only)** Supervisor Level Same as: mfspr rT,1023	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mfpmc1	rT	Move From Performance Monitor Counter 1 **(604[e], 620 only)** Supervisor Level Same as: mfspr rT,953	
mfpmc1rd	rT	Move From Performance Monitor Counter 1/Read Only **(620 only)** Same as: mfspr rT,771	
mfpmc2	rT	Move From Performance Monitor Counter 2 **(604[e], 620 only)** Supervisor Level Same as: mfspr rT,954	
mfpmc2rd	rT	Move From Performance Monitor Counter 2/Read Only **(620 only)** Same as: mfspr rT,772	
mfpmc3	rT	Move From Performance Monitor Counter 3 **(604e only)** Supervisor Level Same as: mfspr rT,957	
mfpmc4	rT	Move From Performance Monitor Counter 4 **(604e only)** Supervisor Level Same as: mfspr rT,958	
mfpvr	rT	Move From Processor Version Register Supervisor Level Same as: mfspr rT,287	
mfrpa	rT	Move From Required Physical Address **(602, 603[e] only)** Supervisor Level Same as: mfspr rT,982	
mfrtcl	rT	Move From Real Time Clock Lower **(601 only)** Same as: mfspr rT,5	"
mfrtcu	rT	Move From Real Time Clock Upper **(601 only)** Same as: mfspr rT,4	"
mfsda	rT	Move From Sampled Data Address Register **(604[e], 620 only)** Supervisor Level Same as: mfspr rT,959	
mfsdr1	rT	Move From Storage Description Register 1 Supervisor Level Same as: mfspr rT,25	
mfsebr	rT	Move From Special Execute Base Register **(602 only)** Supervisor Level Same as: mfspr rT,990	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mfser	rT	Move From Special Execute Register **(602 only)** Supervisor Level Same as: mfspr rT,991	
mfsia	rT	Move From Sampled Instruction Address Register **(604[e], 620 only)** Supervisor Level Same as: mfspr rT,955	
mfsp	rT	Move From Single-Precision Tag Register **(602 only)** Supervisor Level Same as: mfspr rT,1021	
mfspr	rT,SPR	Move From Special Purpose Register Supervisor Level if SPR[0]==1	"
mfsprg	rT,n	Move From Special Purpose Register General n Supervisor Level Same as: mfspr rT,272+n	
mfsr	rT,SR	Move From Segment Register Supervisor Level	"
mfsrin	rT,rB	Move From Segment Register Indirect Supervisor Level	mfsri
mfsrr0	rT	Move From Save/Restore Register 0 Supervisor Level Same as: mfspr rT,26	
mfsrr1	rT	Move From Save/Restore Register 1 Supervisor Level Same as: mfspr rT,27	
mftb	rT,TBR	Move From Time Base (lower) **(not on 601)** Note that 64-bit implementations get all 64 bits with this one instruction Same as: mftb rT,268	
mftbl	rT	Move From Time Base Lower **(not on 601,64-bit)** Same as: mftb rT,268	
mftbu	rT	Move From Time Base Upper **(not on 601,64-bit)** Same as: mftb rT,269	
mftcr	rT	Move From Time Control Register **(602 only)** Supervisor Level Same as: mfspr rT,984	
mfthrm1	rT	Move From Thermal 1 **(750 only)** Supervisor Level Same as : mfspr rT, 1020	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mfthrm2	rT	Move From Thermal 1 **(750 only)** Supervisor Level Same as : mfspr rT,1021	
mfthrm3	rT	Move From Thermal 1 **(750 only)** Supervisor Level Same as : mfspr rT,1022	
mfummcr0	rT	Move From User Mask Register 0 **(750 only)** Same as: mfspr rT,936	
mfummcr1	rT	Move From User Mask Register 1 **(750 only)** Same as: mfspr rT, 940	
mfupmc1	rT	Move From User Performance Monitor Counter 1 **(750 only)** Same as: mfspr rT, 937	
mfupmc2	rT	Move From User Performance Monitor Counter 2 **(750 only)** Same as: mfspr rT, 938	
mfupmc3	rT	Move From User Performance Monitor Counter 4 **(750 only)** Same as: mfspr rT, 941	
mfupmc4	rT	Move From User Performance Monitor Counter 4 **(750 only)** Same as: mfspr rT, 942	
mfusia	rT	Move From User Sampled Instruction Address Register **(750 only)** Same as: mfspr rT, 939	
mfxer	rT	Move From Fixed-Point Exception Register Same as: mfspr rT,1	"
mr[.]	rT,rA	Move Register Same as: or[.] rT,rA,rA	"
mtasr	rS	Move To Address Space Register Supervisor Level Same as: mtspr 280,rS	
mtbatl	n,rS	Move To Block Address Translation Lower n **(601 only)** Supervisor Level Same as: mtspr 529+2n,rS	
mtbatu	n,rS	Move To Block Address Translation Upper n **(601 only)** Supervisor Level Same as: mtspr 528+2n,rS	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mtbuscr	rS	Move To Bus Control & Status Register **(620 only)** Supervisor Level Same as: mtspr 1016,rS	
mtcr	rS	Move To Condition Register Same as: mtcrf 0xff,rS	
mtcrf	FXM,rS	Move To Condition Register Fields	"
mtctr	rS	Move To Count Register Same as: mtspr 9,rS	"
mtdabr	rS	Move To Data Address Breakpoint Register **(601,604[e],620 only)** Supervisor Level Same as: mtspr 1013,rS	
mtdar	rS	Move To Data Address Register Supervisor Level Same as: mtspr 19,rS	
mtdbatl	n,rS	Move To Data Block Address Translation Lower n **(not on 601)** Supervisor Level Same as: mtspr 537+2n,rS	
mtdbatu	n,rS	Move To Data Block Address Translation Upper n **(not on 601)** Supervisor Level Same as: mtspr 536+2n,rS	
mtdec	rS	Move To Decrementer Supervisor Level Same as: mtspr 22,rS	"
mtdsisr	rS	Move To Data Storage Interrupt Status Register Supervisor Level Same as: mtspr 18,rS	
mtear	rS	Move To External Access Register (optional) Supervisor Level Same as: mtspr 282,rS	
mtesasrr	rS	Move To Enable Supervisor Access Save and Restore Register **(602 only)** Supervisor Level Same as: mtspr 987, rS	
mtfpecr	rS	Move To Floating-Point Exception Cause Register (optional) **(not on 601, 602, 603[e], 604[e])** Supervisor Level Same as: mtspr 1022,rS	
mtfsb0[.]	BT	Move To FPSRC Bit a 0	"

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mtfsb1[.]	BT	Move To FPSRC Bit a 1	"
mtfsf[.]	FLM,fB	Move To FPSCR Fields	"
mtfsfi[.]	BFT,U	Move To FPSCR Field Immediate	"
mthid0	rS	Move To Hardware Implementation Dependent 0 **(601, 602, 603[e], 604[e], 620 only)** Supervisor Level Same as: mtspr 1008,rS	
mthid1	rS	Move to Hardware Implementation Dependent 1 **(601, 602, 603e, 604e only)** Supervisor Level Same as: mtspr 1009,rS	
mtiabr	rS	Move To Instruction Address Breakpoint Register **(601, 602, 603[e], 604[e], 620 only)** Supervisor Level Same as: mtspr 1010,rS	
mtibatl	n,rS	Move To Instruction Block Address Translation Lower n **(not on 601)** Supervisor Level Same as: mtspr 529+2n,rS	
mtibatu	n,rS	Move to Instruction Block Address Translation Upper n **(not on 601)** Supervisor Level Same as: mtspr 528+2n,rS	
mtibr	rS	Move To Interrupt Base Register **(602 only)** Supervisor Level Same as: mtspr 986,rS	
mtictc	rS	Move To Instruction Cache-Throttling Control Register **(750 only)**	
mtl2sr	rS	Move To L2 Status Register **(620 only)** Supervisor Level Same as: mtspr 1018,rS	
mtlr	rS	Move To Link Register Same as: mtspr 8,rS	"
mtlt	rS	Move To Integer Tag Register **(602 only)** Supervisor Level Same as: mfspr 1022,rS	
mtmmcr0	rS	Move To Mask Register 0 **(604[e], 620 only)** Supervisor Level Same as: mtspr 952,rS	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mtmmcr1	rS	Move To Mask Register 1 **(604e only)** Supervisor Level Same as: mtspr 956,rS	
mtmq	rS	Move To Multiply Quotient Register **(601 only)** Same as: mtspr 0,rS	"
mtmsr	rS	Move To Machine State Register Supervisor Level	"
mtpir	rS	Move To Processor ID Register (optional) **(not on 602, 603[e])** Supervisor Level Same as: mtspr 1023,rS	
mtpmc1	rS	Move To Performance Monitor Counter 1 **(604[e], 620 only)** Supervisor Level Same as: mtspr 953,rS	
mtpmc2	rS	Move To Performance Monitor Counter 2 **(604[e], 620 only)** Supervisor Level Same as: mtspr 954,rS	
mtpmc3	rS	Move To Performance Monitor Counter 3 **(604e only)** Supervisor Level Same as: mtspr 957,rS	
mtpmc4	rS	Move To Performance Monitor Counter 4 **(604e only)** Supervisor Level Same as: mtspr 958,rS	
mtrpa	rS	Move To Required Physical Address **(602, 603[e] only)** Supervisor Level Same as:mtspr 982,rS	
mtrtcl	rS	Move To Real Time Clock Lower **(601 only)** Same as: mtspr 21,rS	"
mtrtcu	rS	Move To Real Time Clock Upper **(601 only)** Same as: mtspr 20,rS	"
mtsda	rS	Move To Sampled Data Address Register **(604[e], 620 only)** Supervisor Level Same as: mtspr 959,rS	
mtsdr1	rS	Move to Storage Description Register 1 Supervisor Level Same as: mtspr 25,rS	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mtsebr	rS	Move To Special Execute Base Register **(602 only)** Supervisor Level Same as: mtspr 990,rS	
mtser	rS	Move To Special Execute Register **(602 only)** Supervisor Level Same as: mtspr 991,rS	
mtsia	rS	Move To Sampled Instruction Address Register **(604[e], 620 only)** Supervisor Level Same as: mtspr 955,rS	
mtsp	rS	Move To Single-Precision Tag Register **(602 only)** Supervisor Level Same as: mtspr 1021,rS	
mtspr	SPR,rS	Move To Special Purpose Register Supervisor Level if SPR[0]==1	"
mtsprg	n,rS	Move to Special Purpose Register General n Supervisor Level Same as: mtspr 272+n,rS	
mtsr	SR,rS	Move To Segment Register Supervisor Level	"
mtsrd	SR,rS	Move Doubleword To Segment Register Supervisor Level	
mtsrdin or **mtsrind**	rS,rB	Move Doubleword To Segment Register Indirect Supervisor Level	
mtsrin	rS,rB	Move To Segment Register Indirect Supervisor Level	mtsri
mtsrr0	rS	Move to Save/Restore Register 0 Supervisor Level Same as: mtspr 26,rS	
mtsrr1	rS	Move to Save/Restore Register 1 Supervisor Level Same as: mtspr 27,rS	
mttb	rS	Move to Time Base (lower) **(not on 601)** Supervisor Level Same as: mtspr 284,rS	
mttbl	rS	Move to Time Base Lower **(not on 601,64-bit)** Supervisor Level Same as: mtspr 284,rS	

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
mttbu	rS	Move to Time Base Upper **(not on 601,64-bit)** Supervisor Level Same as: mtspr 285,rS	
mttcr	rS	Move To Time Control Register **(602 only)** Supervisor Level Same as: mtspr 984,rS	
mtthrm1	rS	Move To Thermal 1 **(750 only)** Supervisor Level Same as: mtspr 1020,rS	
mtthrm2	rS	Move To Thermal 1 **(750 only)** Supervisor Level Same as: mtspr 1021,rS	
mtthrm3	rS	Move To Thermal 1 **(750 only)** Supervisor Level Same as: mtspr 1022,rS	
mtxer	rS	Move To Fixed-Point Exception Register Same as: mtspr 1,rS	
mul[o][.]	rT,rA,rB	Multiply **(601 only)**	"
mulhd[.]	rT,rA,rB	Multiply High Doubleword	
mulhdu[.]	rT,rA,rB	Multiply High Doubleword Unsigned	
mulhw[.]	rT,rA,rB	Multiply High Word	
mulhwu[.]	rT,rA,rB	Multiply High Word Unsigned	
mulld[o][.]	rT,rA,rB	Multiply Low Doubleword	
mulli	rT,rA,SI	Multiply Low Immediate	muli
mullw[o][.]	rT,rA,rB	Multiply Low Word	muls[o][.]
nabs[o][.]	rT,rA	Negative Absolute Value **(601 only)**	"
nand[.]	rA,rS,rB	Not AND	"
neg[o][.]	rT,rA	Negate	"
nop	-	No Operation Same as: ori r0,r0,0	"
no-op	-	No Operation Same as: ori r0,r0,0	nop
not[.]	rA,rS	NOT Same as: nor[.] rA,rS,rS	
nor[.]	rA,rS,rB	Not OR	"
or[.]	rA,rS,rB	OR	"

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
orc[.]	rA,rS,rB	OR with Complement	"
ori	rA,rS,UI	OR Immediate	oril
oris	rA,rS,UI	OR Immediate Shifted	oriu
rfi	-	Return From Interrupt Supervisor Level	"
rldcl[.]	rA,rS,rB,MB	Rotate Left Doubleword then Clear Left	
rldcr[.]	rA,rS,rB,ME	Rotate Left Doubleword then Clear Right	
rldic[.]	rA,rS,SH,MB	Rotate Left Doubleword Immediate then Clear	
rldicl[.]	rA,rS,SH,MB	Rotate Left Doubleword Immediate then Clear Left	
rldicr[.]	rA,rS,SH,ME	Rotate Left Doubleword Immediate then Clear Right	
rldimi[.]	rA,rS,SH,MB	Rotate Left Doubleword Immediate then Mask Insert	
rlmi[.]	rA,rS,rB,MB[,ME]	Rotate Left Then Mask Insert **(601 only)**	"
rlwimi[.]	rA,rS,SH,MB[,ME]	Rotate Left Word Immediate then Mask Insert If ME is omitted, MB is the mask rather than the beginning bit of the mask	rlimi[.]
rlwinm[.]	rA,rS,SH,MB[,ME]	Rotate Left Word Immediate then AND with Mask If ME is omitted, MB is the mask rather than the beginning bit of the mask	rlinm[.]
rlwnm[.]	rA,rS,rB,MB[,ME]	Rotate Left Word then AND with Mask If ME is omitted, MB is the mask rather than the beginning bit of the mask	rlnm[.]
rotld[.]	rA,rS,rB	Rotate Left Doubleword Same as: rldcl rA,rS,rB,0	
rotldi[.]	rA,rS,n	Rotate Left Doubleword Immediate Same as rldicl rA,rS,n,0	
rotlw[.]	rA,rS,rB	Rotate Left Word Same as: rlwnm[.] rA,rS,rB,0,31	
rotlwi[.]	rA,rS,n	Rotate Left Word Immediate Same as: rlwinm[.] rA,rS,n,0,31	
rotrdi[.]	rA,rS,n	Rotate Right Doubleword Immediate Same as: rldicl rA,rS,64-n,0	
rotrwi[.]	rA,rS,n	Rotate Right Word Immediate Same as: rlwinm[.] rA,rS,32-n,0,31	
rrib[.]	rA,rS,rB	Rotate Right And Insert Bit **(601 only)**	"
sc	-	System Call	svca

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
slbia	-	SLB Invalidate All (optional) Supervisor Level	
slbie	rB	SLB Invalidate Entry (optional) Supervisor Level	
slbiex	rB	SLB Invalidate Entry by Index (optional) **(not on 620)** Supervisor Level	
sld[.]	rA,rS,rB	Shift Left Doubleword	
sldi[.]	rA,rS,n	Shift Left Doubleword Immediate Same as rldicl rS,rS,n,63-n	
sle[.]	rA,rS,rB	Shift Left Extended **(601 only)**	"
sleq[.]	rA,rS,rB	Shift Left Extended with MQ **(601 only)**	"
sliq[.]	rA,rS,SH	Shift Left Immediate with MQ **(601 only)**	"
slliq[.]	rA,rS,SH	Shift Left Long Immediate with MQ **(601 only)**	"
sllq[.]	rA,rS,rB	Shift Left Long with MQ **(601 only)**	"
slq[.]	rA,rS,rB	Shift Left with MQ **(601 only)**	"
slw[.]	rA,rS,rB	Shift Left Word	sl[.]
slwi[.]	rA,rS,n	Shift Left Word Immediate Same as: rlwinm[.] rA,rS,n,0,31-n	sli[.]
srad[.]	rA,rS,rB	Shift Right Algebraic Doubleword	
sradi[.]	rA,rS,SH	Shift Right Algebraic Doubleword Immediate	
sraiq[.]	rA,rS,SH	Shift Right Algebraic Immediate With MQ **(601 only)**	"
sraq[.]	rA,rS,rB	Shift Right Algebraic With MQ **(601 only)**	"
sraw[.]	rA,rS,rB	Shift Right Algebraic Word	sra[.]
srawi[.]	rA,rS,SH	Shift Right Algebraic Word Immediate	srai[.]
srd[.]	rA,rS,rB	Shift Right Doubleword	
srdi[.]	rA,rS,n	Shift Right Doubleword Immediate Same as: rldicl rS,rS,64-n,n	
sre[.]	rA,rS,rB	Shift Right Extended **(601 only)**	"
srea[.]	rA,rS,rB	Shift Right Extended Algebraic **(601 only)**	"
sreq[.]	rA,rS,rB	Shift Right Extended With MQ **(601 only)**	"
sriq[.]	rA,rS,SH	Shift Right Immediate With MQ **(601 only)**	"
srliq[.]	rA,rS,SH	Shift Right Long Immediate With MQ **(601 only)**	"
srlq[.]	rA,rS,rB	Shift Right Long With MQ **(601 only)**	"
srq[.]	rA,rS,rB	Shift Right With MQ **(601 only)**	"

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
srw[.]	rA,rS,rB	Shift Right Word	sr[.]
srwi[.]	rA,rS,n	Shift Right Word Immediate Same as: rlwinm rA,rS,32-n,n,31	sri[.]
stb[u]	rS,D(*rA*)	Store Byte	"
stb[u]x	rS,*rA*,rB	Store Byte Indexed	"
std[u]	rS,D(*rA*)	Store Doubleword	
std[u]x	rS,*rA*,rB	Store Doubleword Indexed	
stdcx.	rS,*rA*,rB	Store Doubleword Conditional Indexed	
stfd[u]	fS,D(*rA*)	Store Floating-Point Double **(not on 602)**	"
stfd[u]x	fS,*rA*,rB	Store Floating-Point Double Indexed **(not on 602)**	"
stfiwx	fS,*rA*,rB	Store Floating-Point as Integer Word Indexed (Optional) **(not on 601)**	
stfs[u]	fS,D(*rA*)	Store Floating-Point Single	"
stfs[u]x	fS,*rA*,rB	Store Floating-Point Single Indexed	"
sth[u]	rS,D(*rA*)	Store Halfword	"
sth[u]x	rS,*rA*,rB	Store Halfword Indexed	"
sthbrx	rS,*rA*,rB	Store Halfword Byte-Reverse Indexed	"
stmw	rS,D(*rA*)	Store Multiple Word	stm
stswi	rS,*rA*,NB	Store String Word Immediate **(not on 602)**	stsi
stswx	rS,*rA*,rB	Store String Word Indexed **(not on 602)**	stsx
stw[u]	rS,D(*rA*)	Store Word	st[u]
stw[u]x	rS,*rA*,rB	Store Word Indexed	st[u]x
stwbrx	rS,*rA*,rB	Store Word Byte-Reverse Indexed	stbrx
stwcx.	rS,*rA*,rB	Store Word Conditional Indexed	
sub[o][.]	rT,rA,rB	Subtract Same as: subf rT,rB,rA	
subc[o][.]	rT,rA,rB	Subtract Carrying Same as: subfc rT,rB,rA	
subf[o][.]	rT,rA,rB	Subtract From	
subfc[o][.]	rT,rA,rB	Subtract From Carrying	sf[o][.]
subfe[o][.]	rT,rA,rB	Subtract From Extended	sfe[o][.]
subfic	rT,rA,SI	Subtract From Immediate Carrying	sfi
subfme[o][.]	rT,rA	Subtract From Minus One Extended	sfme[o][.]

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
subfze[o][.]	rT,rA	Subtract From Zero Extended	sfze[o][.]
subi	rT,*rA*,SI	Subtract Immediate Same as: addi rT,rA,-SI	
subic[.]	rT,rA,SI	Subtract Immediate Carrying Same as: addic[.] rT,rA,-SI	
subis	rT,*rA*,SI	Subtract Immediate Shifted Same as: addis rT,rA,-SI	
sync	-	Synchronize	dcs
td	TO,rA,rB	Trap Double	
tdi	TO,rA,SI	Trap Double Immediate	
tdTO	rA,rB	Trap Double If Condition Same as: td TO,rA,rB	
tdTOi	rA,SI	Trap Double Immediate If Condition Same as: tdi TO,rA,SI	
tlbia	-	TLB Invalidate All (optional) (**not on 601, 602, 603[e], 604[e], 620**) Supervisor Level	
tlbie	rB	TLB Invalidate Entry (optional) Supervisor Level	tlbi
tlbiex	rB	TLB Invalidate Entry by Index (optional) (**not on 601, 602, 603[e], 604[e], 620**) Supervisor Level	
tlbld	rB	TLB Load Data Entry **(602, 603[e] only)** Supervisor Level	
tlbli	rB	TLB Load Instruction Entry **(602, 603[e] only)** Supervisor Level	
tlbsync	-	TLB Synchronize (optional) **(not on 601)** Supervisor Level	
trap	-	Trap Unconditionally Same as: tw 31,0,0	
tw	TO,rA,rB	Trap Word	t
twi	TO,rA,SI	Trap Word Immediate	ti
twTO	rA,rB	Trap Word If Condition Same as: tw TO,rA,rB	tTO
twTOi	rA,SI	Trap Word Immediate If Condition Same as: twi TO,rA,SI	tTOi

Table 3-1. PowerPC Instruction Set (Cont.)

PowerPC Mnemonic	Syntax of Operands	Description	RS/6000 POWER Mnemonic
xor[.]	rA,rS,rB	XOR	"
xori	rA,rS,UI	XOR Immediate	xoril
xoris	rA,rS,UI	XOR Immediate Shifted	xoriu

* A ‘+’ or ‘-’ can be appended to conditional branches to indicate predicted branch taken or predicted branch not taken, respectively. A lower order bit of BO being zero means the default prediction; a one means reverse the default prediction. The defaults are: forward branches are predicted not taken, backwards branches are predicted taken, and jumps through link or count register are predicted not taken.

Condition Codes

Table 3-2. Condition Codes (CC)

CC	Meaning	BO	BI	CC	Meaning	BO	BI
eq	equal	8 bdnzCC 10 bdzCC 12 bCC	2+4*CRF	ne	not equal	0 bdnzCC 2 bdzCC 4 bCC	2+4*CRF
gt	greater than	See eq	1+4*CRF	le	less than or equal	See ne	1+4*CRF
				ng	not greater than	See ne	1+4*CRF
lt	less than	See eq	0+4*CRF	ge	greater than or equal	See ne	0+4*CRF
				nl	not less than	See ne	0+4*CRF
so	summary overflow	See eq	3+4*CRF	ns	not summary overflow	See ne	3+4*CRF
un	unordered	See eq	3+4*CRF	nu	not unordered	See ne	3+4*CRF
z	zero	See eq	See eq	nz	not zero	See ne	See ne

For example, the following two instructions are equivalent:

```
beq crf2,L1
bc 12,10,L1
```

Trap Operand

In the following table, * means unsigned comparison.

Table 3-3. Trap Operand (TO)

TO	Meaning	Operand	TO	Meaning	Operand
eq	equal	4	ne	not equal	24
ge	greater than or equal	12	lge	logical greater than or equal (*)	5
gt	greater than	8	lgt	logical greater than (*)	1
le	less than or equal	20	lle	logical less than or equal (*)	6
lt	less than	16	llt	logical less than (*)	2
ng	not greater than	20	lng	logical not greater than (*)	6
nl	not less than	12	lnl	logical not less than (*)	5

For example, the following two instructions are equivalent:

```
tweq r3,r4
tw   4,r3,r4
```

Operand Abbreviations

Table 3-4. Operand Abbreviations

Abbre-viation	Description
BA	bit number in CR: 0-31
BB	bit number in CR: 0-31
BD	14-bit branch displacement: label
BFT	FPSCR field number, target: 0-7
BFA	FPSCR field number, source: 0-7
BI	bit number in CR: 0-31
BO	conditional branch options: 0-31
BT	bit number in CR or FPSCR, target: 0-31
crfT	condition register field target: crf0-crf7
crfA	condition register field source: crf0-crf7
D	16-bit offset: -32768-32767

Table 3-4. Operand Abbreviations (Cont.)

Abbre-viation	Description
FLM	mask of FPSCR fields: 1-255
fA	floating-point register: f0-f31
fB	floating-point register: f0-f31
fC	floating-point register: f0-f31
fS	floating-point register: f0-f31
fT	floating-point register, target: f0-f31
FXM	mask of CR fields: 1-255
L	precision of fixed-point compare: 0-1
LI	24-bit displacement: label
MB	bit number of first bit of mask: 0-31
ME	bit number of last bit of mask: 0-31
NB	byte count: 0-31 (0 means 32)
rA	general register (If italic, then rA=0 means zero): r0-r31
rB	general register: r0-r31
rS	general register: r0-r31
rT	general register, target: r0-r31
SH	shift amount: 0-31
SI	signed 16-bit integer: -32768-32767
SPR	special purpose register: 0-1023
SR	segment register: 0-15
TBR	time base register: 268-269
TO	trap conditions: 0-31
U	immediate value: 0-15
UI	unsigned 16-bit integer: 0-65535

Special-Purpose Registers

Table 3-5. Special-Purpose Registers

Number	Name	Description
0	MQ	Multiply-Quotient Register **(601 only)**
1	XER	Fixed-Point Exception Register
4	RTCU	Real Time Clock Upper (read only) **(601 only)**
5	RTCL	Real Time Clock Lower (read only) **(601 only)**
6	DEC	Decrementer **(601 only)** This is identical to Special Register 22.
8	LR	Link Register
9	CTR	Count Register
18	DSISR	Data Storage Interrupt Status Register
19	DAR	Data Address Register
20	RTCU	Real Time Clock Upper (write only) **(601 only)**
21	RTCL	Real Time Clock Lower (write only) **(601 only)**
22	DEC	Decrementer
25	SDR1	Storage Description Register 1
26	SRR0	Save/Restore Register 0
27	SRR1	Save/Restore Register 1
272	SPRG0	Special Purpose Register General 0
273	SPRG1	Special Purpose Register General 1
274	SPRG2	Special Purpose Register General 2
275	SPRG3	Special Purpose Register General 3
280	**ASR**	Address Space Register **(620 only)**
282	EAR	External Access Register (optional) **(not on 602)**
284	TBL	Time Base Lower (dest only) **(not on 601)**
285	TBU	Time Base Upper (dest only) **(not on 601,64-bit)**
287	PVR	Processor Version Register (src only)
528	IBAT0U	Instruction Block Address Translation 0 Upper
529	IBAT0L	Instruction Block Address Translation 0 Lower
530	IBAT1U	Instruction Block Address Translation 1 Upper
531	IBAT1L	Instruction Block Address Translation 1 Lower
532	IBAT2U	Instruction Block Address Translation 2 Upper
533	IBAT2L	Instruction Block Address Translation 2 Lower

Table 3-5. Special-Purpose Registers (Cont.)

Number	Name	Description
534	IBAT3U	Instruction Block Address Translation 3 Upper
535	IBAT3L	Instruction Block Address Translation 3 Lower
536	DBAT0U	Data Block Address Translation 0 Upper **(not on 601)**
537	DBAT0L	Data Block Address Translation 0 Lower **(not on 601)**
538	DBAT1U	Data Block Address Translation 1 Upper **(not on 601)**
539	DBAT1L	Data Block Address Translation 1 Lower **(not on 601)**
540	DBAT2U	Data Block Address Translation 2 Upper **(not on 601)**
541	DBAT2L	Data Block Address Translation 2 Lower **(not on 601)**
542	DBAT3U	Data Block Address Translation 3 Upper **(not on 601)**
543	DBAT3L	Data Block Address Translation 4 Lower **(not on 601)**
771	PMC1/RD	Performance Monitor Counter 1/Read Only **(620 only)**
772	PMC2/RD	Performance Monitor Counter 2/Read Only **(620 only)**
779	MMCR0/RD	Mask Register/Read Only **(620 only)**
936	UMMCR0	User Mask Register 0 **(750 only)**
937	UPMC1	Performance Monitor Counter 1 **(750 only)**
938	UPMC2	Performance Monitor Counter 2 **(750 only)**
939	USIA	Sampled Instruction Address Register **(750 only)**
940	UMMCR1	Mask Register 1 **(750 only)**
941	UPMC3	Performance Monitor Counter 3 **(750 only)**
942	UPMC4	Performance Monitor Counter 4 **(750 only)**
952	MMCR0	Mask Register 0 **(604[e], 620, 750 only)**
953	PMC1	Performance Monitor Counter 1 **(604[e], 620, 750 only)**
954	PMC2	Performance Monitor Counter 2 **(604[e], 620, 750 only)**
955	SIA	Sampled Instruction Address Register **(604[e], 620, 750 only)**
956	MMCR1	Mask Register 1 **(604e, 750 only)**
957	PMC3	Performance Monitor Counter 3 **(604e, 750 only)**
958	PMC4	Performance Monitor Counter 4 **(604e, 750 only)**
959	SDA	Sampled Data Address Register **(604[e], 620 only)**
976	DMISS	Data TLB Miss Address (src only) **(602, 603[e] only)**
977	DCMP	Data TLB Compare (src only) **(602, 603[e] only)**
978	HASH1	Primary Hash Address (src only) **(602, 603[e] only)**
979	HASH2	Secondary Hash Address (src only) **(602, 603[e] only)**

Table 3-5. Special-Purpose Registers (Cont.)

Number	Name	Description
980	IMISS	Instruction TLB Miss Address (src only) **(602, 603[e] only)**
981	ICMP	Instruction TLB Compare (src only) **(602, 603[e] only)**
982	RPA	Required Physical Address **(602, 603[e] only)**
984	TCR	Timer Control Register **(602 only)**
986	IBR	Interrupt Base Register **(602 only)**
987	ESASRR	ESA Save/Restore Register **(602 only)**
990	SEBR	Special Execute Base Register **(602 on ly)**
991	SER	Special Execute Register **(602 only)**
1008	HID0	Hardware Implementation Dependent 0 **(601, 603[e], 604[e], 620, 750 only)**
1009	HID1	Hardware Implementation Dependent 1 **(601, 602, 603e, 604e, 750 only)**
1010	IABR	Instruction Address Breakpoint Register **(601, 602, 603[e], 604[e], 620, 750 only)**
1013	DABR	Data Address Breakpoint Register **(601, 604[e], 620, 750 only)**
1016	BUSCSR	Bus Control & Status Register **(620 only)**
1017	L2CR	L2 Control Register **(620, 750 only)**
1018	L2SR	L2 Status Register **(620 only)**
1019	ICTC	Instruction Cache-Throttling Control Register **(750 only)**
1020	THRM1	Thermal 1 **(750 only)**
1021	SP	Single-Precision Tag Register **(602 only)**
1021	THRM2	Thermal 2 **(750 only)**
1022	LT	Integer Tag Register **(602 only)**
1022	THRM3	Thermal 3 **(750 only)**
1022	FPECR	Floating-Point Exception Cause Register (optional) **(not on 601, 602, 603[e], 604[e])**
1023	PIR	Processor ID Register (optional) **(not on 602, 603[e], 750)**

Time Base Registers

Table 3-6. Time Base Registers

Number	Name	Description
268	TBL	Time Base **(not on 601)**
269	TBU	Time Base Upper **(not on 601,64-bit)**

Implementation-Specific and Optional Instructions

Table 3-7. Implementation-Specific and Optional Instructions

Mnemonic	601	602	603[e]	604[e]	620	750
abs[o][.]	✔					
clcs	✔					
clrlsdi[.]					✔	
clrldi[.]					✔	
clrrdi[.]					✔	
cmpd					✔	
cmpdi					✔	
cmpld					✔	
cmpldi					✔	
cntlzd[.]					✔	
div[o][.]	✔					
divd[o][.]					✔	
divdu[o][.]					✔	
divs[o][.]	✔					
doz[o][.]	✔					
dozi	✔					
dsa		✔				
eciwx	✔		✔	✔	✔	✔
ecowx	✔		✔	✔	✔	✔
esa		✔				
extldi[.]					✔	
extrdi[.]					✔	

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
extsw[.]					✔	
fadd[.]	✔		✔	✔	✔	✔
fcfid[.]					✔	
fctid[.]					✔	
fctidz[.]					✔	
fdiv[.]	✔		✔	✔	✔	✔
fmadd[.]	✔		✔	✔	✔	✔
fmsub[.]	✔		✔	✔	✔	✔
fmul[.]	✔		✔	✔	✔	✔
fnmadd[.]	✔		✔	✔	✔	✔
fnmsub[.]	✔		✔	✔	✔	✔
fres[.]		✔	✔	✔	✔	✔
frsp[.]	✔		✔	✔	✔	✔
frsqrte[.]		✔	✔	✔	✔	✔
fsel[.]		✔	✔	✔	✔	✔
fsqrt[s][.]					✔	
fsub[.]	✔		✔	✔	✔	✔
insrdi[.]					✔	
ld[u]					✔	
ld[u][x]					✔	
ldarx					✔	
lfd[u]	✔		✔	✔	✔	✔
lfd[u]x	✔		✔	✔	✔	✔
lscbx[.]	✔					
lswi	✔		✔	✔	✔	✔
lswx	✔		✔	✔	✔	✔
lwa					✔	
lwax					✔	
maskg[.]	✔					
maskir[.]	✔					
mfasr					✔	
mfbatl	✔					
mfbatu	✔					

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
mfbuscr					✔	
mfdabr	✔			✔	✔	✔
mfdbatl		✔	✔	✔	✔	✔
mfdbatu		✔	✔	✔	✔	✔
mfdcmp		✔	✔			
mfdmiss		✔	✔			
mfear	✔		✔	✔	✔	✔
mfesasrr	✔					
mffpecr					✔	
mfhash1		✔	✔			
mfhash2		✔	✔			
mfhid0	✔	✔	✔	✔	✔	✔
mfhid1	✔	✔	**603e**	**604e**		✔
mfiabr	✔	✔	✔	✔	✔	✔
mfibatl		✔	✔	✔	✔	✔
mfibatu		✔	✔	✔	✔	✔
mfibr	✔					
mficmp		✔	✔			
mfimiss		✔	✔			
mfl2cr					✔	
mfl2sr					✔	
mflt		✔				
mfmmcr0				✔	✔	✔
mfmmcr0rd					✔	
mfmmcr1				**604e**		✔
mfmq	✔					
mfpir	✔			✔	✔	
mfpmc1				✔	✔	✔
mfpmc1rd					✔	
mfpmc2				✔	✔	✔
mfpmc2rd					✔	
mfpmc3				**604e**		✔
mfpmc4				**604e**		✔

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
mfrpa		✔	✔			
mfrtcl	✔					
mfrtcu	✔					
mfsda				✔	✔	
mfsebr		✔				
mfser		✔				
mfsia				✔	✔	✔
mfsp		✔				
mftb		✔	✔	✔	✔	✔
mftbl		✔	✔	✔		✔
mftbu		✔	✔	✔		✔
mftcr		✔				
mtasr					✔	
mtbatl	✔					
mtbatu	✔					
mtbuscr					✔	
mtdabr	✔			✔	✔	✔
mtdbatl		✔	✔	✔	✔	✔
mtdbatu		✔	✔	✔	✔	✔
mtear	✔		✔	✔	✔	✔
mtesasrr		✔				
mtfpecr					✔	
mthid0	✔	✔	✔	✔	✔	✔
mthid1	✔	✔	**603e**	**604e**		✔
mtiabr	✔	✔	✔	✔	✔	✔
mtibatl		✔	✔	✔	✔	✔
mtibatu		✔	✔	✔	✔	✔
mtibr		✔				
mtl2cr					✔	
mtl2sr					✔	
mtlt		✔				
mtmmcr0				✔	✔	✔
mtmmcr1				**604e**		✔

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
mtmq	✔					
mtpir	✔			✔	✔	
mtpmc1				✔	✔	✔
mtpmc2				✔	✔	✔
mtpmc3				**604e**		✔
mtpmc4				**604e**		✔
mtrpa		✔	✔			
mtrtcl	✔					
mtrtcu	✔					
mtsda				✔	✔	
mtsebr		✔				
mtser		✔				
mtsia				✔	✔	✔
mtsp		✔				
mtsrd					✔	
mtsrdin or **mtsrind**					✔	
mttb		✔	✔	✔	✔	✔
mttbl		✔	✔	✔		✔
mttbu		✔	✔	✔		✔
mttcr		✔				
mul[o][.]	✔					
mulhd[.]					✔	
mulhdu[.]					✔	
mulld[o][.]					✔	
nabs[o][.]	✔					
rldcl[.]					✔	
rldcr[.]					✔	
rldic[.]					✔	
rldicl[.]					✔	
rldicr[.]					✔	
rldimi[.]					✔	
rlmi[.]	✔					

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
rotld[.]					✓	
rotldi[.]					✓	
rotrdi[.]					✓	
rrib[.]	✓					
slbia					✓	
slbie					✓	
slbiex						
sld[.]					✓	
sldi[.]					✓	
sle[.]	✓					
sleq[.]	✓					
sliq[.]	✓					
slliq[.]	✓					
sllq[.]	✓					
slq[.]	✓					
srad[.]					✓	
sradi[.]					✓	
sraiq[.]	✓					
sraq[.]	✓					
srd[.]					✓	
srdi[.]					✓	
sre[.]	✓					
srea[.]	✓					
sreq[.]	✓					
sriq[.]	✓					
srliq[.]	✓					
srlq[.]	✓					
srq[.]	✓					
std[u]					✓	
std[u]x					✓	
stdcx.					✓	
stfd[u]	✓		✓	✓	✓	✓
stfd[u]x	✓		✓	✓	✓	✓

Table 3-7. Implementation-Specific and Optional Instructions (Cont.)

Mnemonic	601	602	603[e]	604[e]	620	750
stfiwx		✔	✔	✔	✔	✔
stswi	✔		✔	✔	✔	✔
stswx	✔		✔	✔	✔	✔
td					✔	
tdi					✔	
tdTO					✔	
tdTOi					✔	
tlbia						
tlbie	✔	✔	✔	✔	✔	✔
tlbiex						
tlbld		✔	✔			
tlbli		✔	✔			
tlbsync		✔	✔	✔	✔	✔

The Concurrent C compiler provides a number of intrinsic functions to access PowerPC instructions that are not normally generated by C code. See the Concurrent C Reference Manual for details on enabling intrinsics. The following table gives pseudo-prototypes and short descriptions of the provided intrinsics.

The user might infer the existence of some additional intrinsics. However, these intrinsics are not guaranteed to behave as expected and should not be used.

The compiler will generate warnings for any intrinsic inconsistent with the **`Qtarget=`***architecture* option.

Table 3-8. Compiler Intrinsics

Intrinsic	Description
`FRT=(double)`**`__compose_double`**`( int uw,int lw)`	Generate a double-precision floating-point constant given the bit patterns of the two words
`FRT=(float)`**`__compose_float`**`(int w)`	Generate a single-precision floating-point constant given the bit pattern of the word
`__get_fpscr``(double *RA)`	Do an `mffs` and store it in a memory location pointed to by `RA` without modifying any floating point registers
`RT=(unsigned int)`**`__get_thread_reg`**`()`	Get value of the thread register

Table 3-8. Compiler Intrinsics (Cont.)

Intrinsic	Description
`RT=(unsigned int)`**`__inst_clcs`**`(int RA)`	`clcs RA`
`RA=(int)`**`__inst_cntlzw`**`(int RS)`	`cntlzw RA,RS`
`__inst_dcbf``(void *RA,int RB)`	`dcbf  RA,RB`
`__inst_dcbi``(void *RA,int RB)`	`dcbi  RA,RB`
`__inst_dcbst``(void *RA,int RB)`	`dcbst RA,RB`
`__inst_dcbt``(void *RA,int RB)`	`dcbt  RA,RB`
`__inst_dcbtst``(void *RA,int RB)`	`dcbtst RA,RB`
`__inst_dcbz``(void *RA,int RB)`	`dcbz  RA,RB`
`__inst_dsa``(void)`	`dsa`
`RT=(int)`**`__inst_eciwx`**`(int *RA,int RB)`	`eciwx RT,RA,RB`
`__inst_ecowx``(int RS,int *RA,int RB)`	`ecowx RS,RA,RB`
`__inst_eieio``(void)`	`eieio`
`__inst_esa``(void)`	`esa`
`FRT=(float)`**`__inst_fres`**`(double FRB)`	`fres  FRT,FRB`
`FRT=(double)`**`__inst_frsqrte`**`( double FRB)`	`frsqrte FRT,FRB`
`FRT=(double)`**`__inst_fsel`**`(double FRA, double FRC, double FRB)`	`fsel  FRT,FRA,FRC,FRB`
`__inst_icbi``(void *RA,int RB)`	`icbi  RA,RB`
`__inst_isync``(void)`	`isync`
`RT=(short)`**`__inst_lhbrx`**`(short *RA, int RB)`	`lhbrx RT,RA,RB`
`RT=(int)`**`__inst_lwarx`**`(int *RA,int RB)`	`lwarx RT,RA,RB`
`RT=(int)`**`__inst_lwbrx`**`(int *RA,int RB)`	`lwbrx RT,RA,RB`
`RT=(int)`**`__inst_maskg`**`(int RA,int RB)`	`maskg RT,RA,RB`
`RT=(int)`**`__inst_maskir`**`(int RT,int RA, int RB)`	`maskir RT,RA,RB`
`FRT=(double)`**`__inst_mffs`**`(void)`	`mffs  FRT`
`RT=(int)`**`__inst_mfmsr`**`(void)`	`mfmsr RT`
`RT=(int)`**`__inst_mfspr`**`(int spr)`	`mfspr  RT,spr` `spr` must be an integer constant. Extended mnemonics are generated for appropriate register numbers.
`RT=(int)`**`__inst_mfsr`**`(int sr)`	`mfsr  RT,sr` `sr` must be an integer constant

Table 3-8. Compiler Intrinsics (Cont.)

Intrinsic	Description
`RT=(int)`**`__inst_mfsrin`**`(void *RB)`	`mfsrin RT,RB`
`RT=(int)`**`__inst_mftbl`**`(void)`	`mftb  RT,268`
`RT=(int)`**`__inst_mftbu`**`(void)`	`mftb  RT,269`
`__inst_mtfsb0``(int bit)`	`mtfsb0 bit` `bit` must be an integer constant
`__inst_mtfsb1``(int bit)`	`mtfsb1 bit` `bit` must be an integer constant
`__inst_mtfsf``(int mask,double FRB)`	`mtfsf mask,FRB` `mask` must be an integer constant
`__inst_mtfsfi``(int n,unsigned u)`	`mtfsfi n,u` `n` and `u` must be integer constants
`__inst_mtmsr``(int RS)`	`mtmsr RS`
`__inst_mtspr``(int spr,int RS)`	`mtspr spr,RS` `spr` must be an integer constant. Extended mnemonics are generated for appropriate register numbers.
`__inst_mtsr``(int sr,int RS)`	`mtsr sr,RS` `sr` must be an integer constant
`__inst_mtsrin``(int RS,void *RB)`	`mtsrin RS,RB`
`__inst_nop``()`	`ori   r0,r0,0`
`__inst_rfi``()`	`rfi`
`__inst_sc``()`	`sc`
`__inst_sthbrx``(short RS,void *RA, int RB)`	`sthbrx RS,RA,RB`
`__inst_stwbrx``(int RS,void *RA, int RB)`	`stwbrx RS,RA,RB`

`RT=(int)`**`__inst_stwcx_`**`(int RS, void *RA,int RB)`	`stwcx. RS,RA,RB`
	`mfcr   RT`
	`rlwinm RT,RT,3,31,31`

`__inst_sync``(void)`	`sync`
`__inst_tlbia``(void)`	`tlbia`
`__inst_tlbie``(void *RB)`	`tlbie RB`
`__inst_tlbld``(void *RB)`	`tlbld RB`
`__inst_tlbli``(void *RB)`	`rlbli RB`

Table 3-8. Compiler Intrinsics (Cont.)

Intrinsic	Description
`__`**`inst_tlbsync`**`(void)`	`tlbsync`
`__`**`inst_tw`**`(int to,int RA, int RB)`	`tw    to,RA,RB`
`__`**`inst_twi`**`(int to,int RA,int si)`	`twi   to,RA,si`
`RT=(unsigned int)` `__`**`ref_double_first_half`**`(double FA)`	Obtain the bit pattern of the first word of a double-precision floating-point value as an integer
`RT=(unsigned int)` `__`**`ref_double_second_half`**`(double FA)`	Obtain the bit pattern of the second word of a double-precision floating-point value as an integer
`RT=(unsigned int)` `__`**`ref_float_as_uint`**`(float FA)`	Obtain the bit pattern of a single-precision floating-point value as an integer
`RT=(unsigned_int)`**`__rot`**`(` `unsigned int RA,int RB)`	Rotate RA by RB bits
`__`**`set_fpscr`**`(double *RA)`	Do a `mtfsf 0xff` from a memory location pointed to by `RA` without modifying any floating point registers.
`__`**`set_thread_reg`**`(unsigned int RA)`	Set the thread register to a value
`RT=(int)`**`abs`**`(int RA)`	`abs RT,RA` Generate an functionally equivalent code sequence on implementations without an `abs` instruction.
`FT=(float or double)` **`fabs`**`((float or double)FB)`	`fabs FT,RB`
`FT=(double)`**`pow`**`(double FA, double FB)`	Generate code to raise to a power
`FT=(float)`**`powf`**`(float FA, float FB)`	Sngle-precision version of **`pow`**
`FT=(double)`**`sqrt`**`(double FB)`	`fsqrt FT,FB` Generate function call for implementations without an `fsqrt` instruction.
`FT=(single)`**`sqrtf`**`(single FB)`	`fsqrts FT,FB` Generate function call for implementations without an `fsqrts` instruction.

4
Link Editor and Linking

4 Link Editor and Linking

Introduction

Linking is the process of combining object files to produce an executable or another object file. Linking may be done statically or dynamically.

The **ld** command is the static linker, often referred to as the *link editor*. The inputs to **ld** are relocatable object files produced by a compiler, by the assembler, or by a previous invocation of the link editor. The link editor combines these object files to form either a relocatable or an absolute (in other words, executable) object file.

There is no system command which performs dynamic linking. Dynamic linking is performed by user code during the execution of a program.

The link editor supports a command language that allows you to control the **ld** process with great flexibility and precision. Most users, however, do not require the degree of flexibility provided by the command language. In fact, it is usually best to allow the link editor to produce its own layout and perform its own allocation of program resources. The detailed command language supports the ability to:

- Specify the memory configuration of the program
- Combine object file sections in particular fashions
- Bind the files to specific addresses or portions of memory
- Define or redefine global symbols at link edit time

Using the Link Editor

The link editor is invoked as follows.

ld [*options***]** *files*

Files passed to the link editor are object files, libraries containing object files, or text source files containing **ld** directives. The link editor uses the "magic number" (the first two bytes of the file) to determine the file type.

The following options are recognized by **ld**.

-a	In static mode only, produce an executable object file; give errors for undefined references. This is the default behavior for static mode. **-a** may not be used with the **-r** option.

-b In dynamic mode only, when creating an executable, do not do special processing for relocations that reference symbols in shared objects. Without the **-b** option, the link editor will create special position-independent relocations for references to functions defined in shared objects and will arrange for data objects defined in shared objects to be copied into the memory image of the executable by the dynamic linker at run time. With the **-b** option, the output code may be more efficient, but it will be less sharable.

-d{y|n } When **-dy**, the default (if `STATIC_LINK` is not set) is specified, use dynamic linking; when **-dn** is specified, use static linking.

-e *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*.

-h *name* In dynamic mode only, when building a shared object, record *name* in the object's dynamic section. *name* will be recorded in executables that are linked with this object rather than the object's system file name. Accordingly, *name* will be used by the dynamic linker as the name of the shared object to search for at run time.

-l*x* Search a library **lib***x***.so** or **lib***x***.a**. **lib***x***.so** and **lib***x***.a** are the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the **-Bstatic** option is in effect, **ld** searches each directory specified in the library search path for a file **lib***x***.so** or **lib***x***.a**. The directory search stops at the first directory containing either. **ld** chooses the file ending in **.so** if **-l***x* expands to two files whose names are of the form **lib***x***.so** and **lib***x***.a**. If no **lib***x***.so** is found, then **ld** accepts **lib***x***.a**. In static mode, or when the **-Bstatic** option is in effect, **ld** selects only the file ending in **.a**. A library is searched when its name is encountered, so the placement of **-l** is significant. By default, libraries are located in **/lib**, **/usr/lib**, and **/usr/ccs/lib**.

-m Produce a map or listing of the input/output sections on the standard output.

-o *outfile* Produce an output object file by the name *outfile*. The name of the default object file is **a.out**.

-r Combine relocatable object files to produce one relocatable object file. **ld** will not complain about unresolved references. This option cannot be used in dynamic mode or with **-a**. Relocation entries are retained in the output file so that it can become an input file in a subsequent **ld** run.

-s Strip symbolic information from the output file. Debug and line information and their associated relocation entries will be removed. Except for relocatable files or shared objects, the symbol table and string table sections will also be removed from the output object file. Relocation entries will not be saved when this option is used.

`-t`
Turn off the warning about multiply-defined symbols that are not the same size.

`-u` *symname*
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that will define the symbol.

`-v`
Same as **`-V`**.

`-x`
Do not preserve local symbols with type `STT_NOTYPE`. This option saves some space in the output file.

`-x1`
Produce a pseudo-cross reference listing on the standard output. Each file and archive library is examined, and all external symbols are listed along with the names of the object files which define and/or reference the symbols. An executable output file is not produced.

`-zdefs`
Force a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.

`-z{lowzeroes|lowzeros`}
Support dereferencing of null pointers. The link editor creates a segment at addresses 0 (inclusive) through 0x1000 (exclusive), consisting entirely of read-only zeroes.

`-znodefs`
Allow undefined symbols. This is the default when building a shared object. It may be used when building an executable in dynamic mode and linking with a shared object that has unresolved references in routines not used by that executable. This option should be used with caution.

`-ztext`
In dynamic mode only, force a fatal error if any relocations against non-writable, allocatable sections remain.

`-Bbind_now`
In dynamic mode only, this option adds a *DT_BIND_NOW* entry to the *dynamic* section of the output file. This entry instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of *DT_BIND_NOW* takes precedence over a directive to use lazy binding for this object when specified through the environment or via **`dlopen`**.

`-B{dynamic|static`}
Options governing library inclusion. **`-Bdynamic`** is valid in dynamic mode only. These options may be specified any number of times on the command line as toggles: if the **`-Bstatic`** option is given, no shared objects will be accepted until **`-Bdynamic`** is seen. See also the **`-l`** option.

-Bexport[=*list***|:***filename***]**

-Bhide[=*list***|:***filename***]**

list is a comma-separated sequence of symbol names. *filename* contains a list of symbol names, one symbol name per line. Lines beginning with a # character and blank lines are ignored.

Normally, when building a shared object, **ld** makes all global and weak names defined in the shared object visible outside the object itself (exported). When building an executable, it makes visible only those names used by the shared objects with which the executable is linked. All other names are hidden. This behavior can be modified with **-Bhide** and **-Bexport**.

When building a shared object, **-Bexport** is the default. All global and weak definitions are exported. **-Bexport** with a set of symbol names instructs **ld** to hide all global and weak definitions, except those in the specified set. **-Bhide** means to hide all global and weak definitions. **-Bhide** with a set of symbol names means to export all global and weak definitions, except for those in the set of names.

When building an executable, **-Bhide** is the default. Only those names referenced by the shared objects with which the executable is linked are exported. **-Bhide** with a set of symbol names instructs **ld** to export all global and weak definitions, except those in the specified set. Names in a **-Bhide** list that are referenced by the shared objects with which the executable is linked, are ignored, that is, they are exported. **-Bexpor**t means to export all global and weak definitions. **-Bexport** with a set of symbol names means to hide all global and weak definitions except those in the set of names and those referenced by the shared objects with which the executable is linked.

If **-Bhide** and -**Bexport** are used together, one of the options must contain a set of symbol names and the other must not. In this case, the option without the symbol set is ignored. Neither **-Bhide** nor **-Bexport** may be used with **-dn**.

-Bsortbss All uninitialized global variables within a module will be assigned contiguous addresses.

-Bsymbolic[=*list***|:***filename***]**

list is a comma-separated sequence of symbol names. *filename* contains a list of symbol names, one symbol name per line. Lines beginning with a # character and blank lines are ignored.

When building a shared object, if a definition for a named symbol exists, bind all references to the named symbol to that definition. If no list of symbols is provided, bind all references to symbols to definitions that are available; **ld** will issue warnings for undefined symbols unless **-z defs** overrides.

Normally, references to global symbols within shared objects are not bound until run time, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition.

-G — In dynamic mode only, produce a shared object. Undefined symbols are allowed unless the **-z defs** option is specified.

-I *name* — When building an executable, use *name* as the path name of the interpreter to be written into the program header. The default in static mode is no interpreter; in dynamic mode, the default is the name of the dynamic linker, **/usr/lib/libc.so.1**. Either case may be overridden by **-I**. **exec** will load this interpreter when it loads **a.out** and will pass control to the interpreter rather than to **a.out** directly.

-L *path* — Add *path* to the library search directories. **ld** searches for libraries first in any directories specified with **-L** options, then in the standard directories. This option is effective only if it precedes the **-l** option on the command line.

-M *mapfile* — Read *mapfile* as a text file of directives to **ld**. Because these directives change the shape of the output file created by **ld**, use of this option is strongly discouraged.

-O *args* — Invoke the **analyze(1)** tool to perform a static performance analysis, to produce an optimized program, or to produce a profiling program. If *args* begins with a hyphen, the system **analyze(1)** tool is used, and *args* is passed to it. If *args* does not begin with a hyphen, then the first field is considered to be the name of an alternative **analyze(1)** tool, and the remainder of *args* is passed to it.

-QAda — Issue a warning if a user object file contains a global definition or reference of `errno`. Also, set the `EF_PPC_ADA` flag in the program's ELF header.

-QABI — Suppress the output of pointer arrays to tdesc information.

-Qanalyze_patch_size=*size*
Set the amount of patch space reserved for **analyze(1)** profiling to *size*. By default, the reserved size of the patch space is ten times the size of the program's **.text** section. This option is used to change the amount reserved.

-QBSS — Force undefined externals with a positive size into the **.bss** section, even when the **-r** option is used.

-Q{**dynamic|static**}
Same as **-B** {**dynamic|static**}.

-Qfpcr=*value* — Set appropriate fields in the vendor section so that the floating-point control register (`fpscr`) is initialized, on program start-up, to *value*. By default, the `fpscr` specifies the round-to-nearest floating-point rounding mode and the enabling of the floating-point reserved operand, divide-by-zero, and over-flow exceptions. Use of this option effects an override of the default setting of `fpscr` at program start up.

-Qfpexcept=*value* Set appropriate fields in the vendor section so that the machine state register (msr) is initialized, on program start-up, to indicate the kind of floating-point exceptions that can be taken. *value* can be **imprecise** (floating-point exceptions are imprecise and non-recoverable), **precise** (floating-point exceptions are precise and recoverable), or **disabled** (floating-point exceptions do not occur). The default mode is **imprecise.**

If any input module contains floating-point code that is executed speculatively (see chapter 20), the executable program should be link edited with the **-Qfpexcept=disabled** option. Without this option, floating-point exceptions could be raised erroneously.

-QG Same as **-G**.

-QGOTP_TO_GOT Implicitly convert GOTP relocation to GOT relocation during link editing. This permits the static linking of files that have been built to be link edited into shared objects. For improved performance, however, these object files should be rebuilt without the compile-time -**ZPIC** option

-QLD_RUN_PATH=*file*

Accept from *file* a list of library search directories for the dynamic linker. The list is specified as it would be for the `LD_RUN_PATH` environment variable. This option overrides use of the environment variable and is useful when the list is too long for the environment variable. See the discussion of library search directories later in this section.

-Qload=*file* Accept a list of input object files, shared objects, and archives from *file*. This is useful when the list would be too long for the **ld** invocation line.

-Qmult_archive Perform multiple passes over the list of archive libraries to satisfy unresolved symbol references. Each pass examines the archive libraries in the order in which they appear on the invocation line. Without this option, only one pass is made over the list of archive libraries.

-Qno_vendor_reloc

Do not output relocation information in the vendor section of the object file for use by the **analyze(1)** tool. By default, this relocation information is output. This option cannot be used with the **-O** option.

-Qnotdesc Suppress the production of tdesc information.

-Qsearch_order When performing multiple passes over the list of archive libraries to satisfy unresolved symbol references, do not search for unresolved references detected in the current pass until the next pass. This option implies the **-Qmult_archive** option.

-Qsmall_memory By default, the link editor allocates the data space of the program beginning at address segment 3, allowing programs to use up to several address segments of memory for their data space. With this option, the link editor allocates the data space only in address segment 2.

-Qstandard_fortran_common
By default, the link editor checks for and properly handles certain nonstandard Fortran common block constructs, but at the expense of increased link time. Use of this option reduces link time, but it presumes that all Fortran common blocks are strictly standard conforming. Unexpected results could be obtained if this option is used and nonstandard Fortran common block extensions are present.

-Qsymbolic Same as **-Bsymbolic**.

-Qwarn_mult_init Warns if a Fortran common block is multiply initialized. If a particular byte in the common block is multiply initialized, the last initialized value of the byte is selected. Without this option, no warning is produced, and all initialization values of the byte are `OR`'ed together.

-Q{**y**|**n**} Same as **-d**{**y**|**n**}.

-V Output a message giving information about the version of **ld** being used.

-X Do not look in alternate search paths for libraries. An error message will be generated if the libraries cannot be located in the specified search path(s).

-YP, *dirlist* Change the default directories used for finding libraries. *dirlist* is a colon-separated path list.

The environment variable `LD_LIBRARY_PATH` may be used to specify library search directories. In the most general case, it will contain two directory lists separated by a semicolon:

dirlist1;dirlist2

If **ld** is called with any number of occurrences of **-L**, as in

ld ... -L*path1* **...-L***pathn* **...**

then the search path ordering is

dirlist1 path1 . . . pathn dirlist2 LIBPATH

`LD_LIBRARY_PATH` is also used to specify library search directories to the dynamic linker at run time. That is, if `LD_LIBRARY_PATH` exists in the environment, the dynamic linker will search the directories named in it, before its default directory, for shared objects to be linked with the program at execution.

The environment variable `LD_RUN_PATH`, containing a directory list, may also be used to specify library search directories to the dynamic linker. If present and not null, it is passed to the dynamic linker by **ld** via data stored in the output object file.

Basics of Linking

If any argument to **`ld`** is a library, it is searched exactly once (by default) at the point it is encountered in the argument list. The library may be either a relocatable archive or a shared object. For an *archive library*, only those routines defining an unresolved external reference are loaded. The archive library symbol table [see **`ar(4)`**] is searched sequentially with as many passes as are necessary to resolve external references that can be satisfied by library members. Thus, the ordering of members in the library is functionally unimportant, unless there exist multiple library members defining the same external symbol. A *shared object* consists of a single entity all of whose references must be resolved within the executable being built or within other shared objects with which it is linked.

NOTE

Because we try to cover the widest possible audience in this section, it may provide more background than many users will need to link their programs with a C language library. If you are interested only in the how-to, and are comfortable with a purely formal presentation that scants motivation and background alike, you may want to skip to “Quick-Reference Guide” on page 4-35.

Link editing refers to the process in which a symbol referenced in one module of your program is connected with its definition in another--for example, the process by which the symbol `printf()` in an example source file **`hello.c`** is connected with its definition in the standard C library.

The link editor uses two models of linking, static or dynamic, as governed by the **`-d`** option or by the presence of the `STATIC_LINK` environment variable. If this environment variable is not set, then dynamic linking is the model used, unless overridden by the **`-dn`** option. If this environment variable is set, then static linking is the model used, unless overridden by the **`-dy`** option.

Whichever link editing model you choose, static or dynamic, the link editor will search each module of your program, including any libraries you have used, for definitions of undefined external symbols in the other modules. If it does not find a definition for a symbol, the link editor will report an error by default, and fail to create an executable program. (Multiply-defined symbols are treated differently, however, under each approach. For details, see “Multiply-Defined Symbols” on page 4-22.) The principal difference between static and dynamic linking lies in what happens after this search is completed:

- Under static linking, copies of the archive library object files that satisfy still unresolved external references in your program are incorporated in your executable at link time. External references in your program are connected with their definitions--assigned addresses in memory--when the executable is created.
- Under dynamic linking, the contents of a shared object are mapped into the virtual address space of your process at run time. External references in

your program are connected with their definitions when the program is executed.

In this section, we'll examine the link editing process in detail. We'll start with the default arrangement, and with the basics of linking your program with the standard libraries supplied by the C compilation system. Later, we'll discuss the implementation of the dynamic linking mechanism, and look at some coding guidelines and maintenance tips for shared library development. Throughout the discussion, we'll consider the reasons why you might prefer dynamic to static linking. These are, briefly:

- Dynamically linked programs save disk storage and system process memory by sharing library code at run time.
- Dynamically linked code can be fixed or enhanced without having to relink applications that depend on it.

Default Arrangement

We stated earlier that the default **cc** command line

```
cc file1.c file2.c file3.c
```

would create object files corresponding to each of your source files, and link them with each other to create an executable program. These object files are called relocatable object files because they contain references to symbols that have not yet been connected with their definitions--have not yet been assigned addresses in memory.

We also suggested that this command line would arrange for the standard C library functions that you have called in your program to be linked with your executable automatically. The standard C library is, in this default arrangement, a shared object called **`libc.so`**, which means that the functions you have called will be linked with your program at run time. (There are some exceptions. A number of C library functions have been left out of **`libc.so`** by design. If you use one of these functions in your program, the code for the function will be incorporated in your executable at link time. That is, the function will still be automatically linked with your program, only statically rather than dynamically.) The standard C library contains the system calls described in Section 2 man pages, and the C language functions described in Section 3, Subsections 3C and 3S man pages.

Now let's look at the formal basis for this arrangement:

1. By convention, shared objects, or dynamically linked libraries, are designated by the prefix **`lib`** and the suffix **`.so`**; archives, or statically linked libraries, are designated by the prefix **`lib`** and the suffix **`.a`**. **`libc.so`**, then, is the shared object version of the standard C library; **`libc.a`** is the archive version.
2. These conventions are recognized, in turn, by the **`-l`** option to the **`cc`** command. That is,

   ```
   cc file1.c file2.c file3.c -lx
   ```

   directs the link editor to search the shared object **`lib`***x***`.so`** or the archive library **`lib`***x***`.a`**. The **`cc`** command automatically passes **`-lc`** to the link editor.

3. By default, the link editor chooses the shared object implementation of a library, **lib***x***.so**, in preference to the archive library implementation, **lib***x***.a**, in the same directory.
4. By default, the link editor searches for libraries in the standard places on the system, **/usr/lib** and **/lib**, in that order.

Adding it up, we can say, more exactly than before, that the default **cc** command line will direct the link editor to search **/usr/lib/libc.so** rather than its archive library counterpart. We'll look at each of the items that make up the default in more detail below.

libc.so is, with one exception, the only shared object library supplied by the C compilation system. (The exception, **libdl.so**, is used with the programming interface to the dynamic linking mechanism described later. Other shared object libraries are supplied with the operating system, and usually are kept in the standard places.) In the next subsection, we'll show you how to link your program with the archive version of **libc** to avoid the dynamic linking default. Of course, you can link your program with libraries that perform other tasks as well. Finally, you can create your own shared objects and archive libraries. We'll show you the mechanics of doing that below.

The default arrangement, then, is this: the **cc** command creates and then links relocatable object files to generate an executable program, then arranges for the executable to be linked with the shared C library at run time. If you are satisfied with this arrangement, you need make no other provision for link editing on the **cc** command line.

Linking with Standard Libraries

libc.so is a single object file that contains the code for every function in the shared C library. When you call a function in that library, and dynamically link your program with it, the entire contents of **libc.so** are mapped into the virtual address space of your process at run time.

Archive libraries are configured differently. Each function, or small group of related functions (typically, the related functions that you will sometimes find on the same manual page), is stored in its own object file. These object files are then collected in archives that are searched by the link editor when you specify the necessary options on the **cc** command line. The link editor makes available to your program only the object files in these archives that contain a function you have called in your program. You create a shared object library by specifying the **-Zlink=so** option to the compiler:

As noted, **libc.a** is the archive version of the standard C library. The **cc** command will automatically direct the link editor to search **libc.a** if you turn off the dynamic linking default with the **-Zlink=static** option:

```
cc -Zlink=static file1.c file2.c file3.c
```

Copies of the object files in **libc.a** that resolve still unresolved external references in your program will be incorporated in your executable at link time.

If you need to point the link editor to standard libraries that are not searched automatically, you specify the **-l** option explicitly on the **cc** command line. As we have seen, **-l***x* directs the link editor to search the shared object **lib***x***.so** or the archive library **lib***x***.a**. So if your program calls the function `sin()`, for example, in the standard math library **`libm`**, the command

```
cc file1.c file2.c file3.c -lm
```

will direct the link editor to search for **`/usr/lib/libm.so`**, and if it does not find it, **`/lib/libm.a`**, to satisfy references to `sin()` in your program. Because the compilation system supplies shared object versions only of **`libc`** and **`libdl`**, the above command will direct the link editor to search **`libm.a`** unless you have installed a shared object version of **`libm`** in the standard place. Note that because we did not turn off the dynamic linking default with the **`-Zlink=static`** option, the above command will direct the link editor to search **`libc.so`** rather than **`libc.a`**. You would use the same command with the **`-Zlink=static`** option to link your program statically with **`libm.a`** and **`libc.a`**. The contents of **`libm`** are described in Chapter 16 (“Run-Time Libraries”).

Note, finally, that because the link editor searches an archive library only to resolve undefined external references it has previously seen, the placement of the **`-l`** option on the **`cc`** command line is important. That is, the command

```
cc -Zlink=static file1.c -lm file2.c file3.c
```

will direct the link editor to search **`libm.a`** only for definitions that satisfy still unresolved external references in **`file1.c`**. As a rule, then, it’s best to put **`-l`** at the end of the command line.

Creating and Linking with Archive and Shared Object Libraries

In this subsection we describe the basic mechanisms by which archives and shared objects are built. The idea is to give you some sense of where these libraries come from, as a basis for understanding how they are implemented and linked with your programs. Of course, if you are developing a library, you will need to know the material in this subsection. Even if you are not, it should prove a useful introduction to the subsequent discussion.

The following commands

```
cc -c function1.c function2.c function3.c
ar -r libfoo.a function1.o function2.o function3.o
```

will create an archive library, **`libfoo.a`**, that consists of the named object files. (Check the **`ar(1)`** manual page for details of usage.) When you use the **`-l`** option to link your program with **`libfoo.a`**

```
cc -L dir file1.c file2.c file3.c -lfoo
```

the link editor will incorporate in your executable only the object files in this archive that contain a function you have called in your program. Note, again, that because we did not turn off the dynamic linking default with the **`-Zlink=static`** option, the above command will direct the link editor to search **`libc.so`** as well as **`libfoo.a`**. We’ll look at the directory search option--represented in the above command line by **`-L`** *dir*--in the next subsection. For now it’s enough to note that you use it to point the link editor to the directory in which your library is stored.

As mentioned earlier, you create a shared object library by specifying the **`-Zlink=so`** option to the compiler:

```
cc -Zlink=so -o libfoo.so function1.o function2.o \
    function3.o
```

That command will create the shared object **`libfoo.so`** consisting of the object code for the functions contained in the named files. (We are deferring for the moment a discussion of a compiler option, **`-ZPIC`**, that you should use in creating a shared object. For that discussion, see "Implementation" on page 4-17.) When you use the **`-l`** option to link your program with **`libfoo.so`**

```
cc -L dir file1.c file2.c file3.c -lfoo
```

the link editor will record in your executable the name of the shared object and a small amount of bookkeeping information for use by the system at run time. Another component of the system--the dynamic linker--does the actual linking.

A number of things are worth pointing out here. First, because shared object code is not copied into your executable object file at link time, a dynamically linked executable normally will use less disk space than a statically linked executable. For the same reason, shared object code can be changed without breaking executables that depend on it. In other words, even if the shared C library were enhanced in the future, you would not have to relink programs that depended on it (as long as the enhancements were compatible with your code; see "Checking for Run-Time Compatibility" on page 4-16). The dynamic linker would simply use the definitions in the new version of the library to resolve external references in your executables at run time.

Second, we specified the name of the shared object that we wanted to be created under the **`-Zlink=so`** option. Of course, you don't have to do it the way we did. The following command, for example, will create a shared object called **`a.out`**:

```
cc -Zlink=so function1.o function2.o function3.o
```

You can then rename the shared object:

```
mv a.out libfoo.so
```

As noted, you use the **`lib`** prefix and the **`.so`** suffix because they are conventions recognized by **`-l`**, just as are **`lib`** and **`.a`** for archive libraries. So while it is legitimate to create a shared object that does not follow the naming convention, and to link it with your program

```
cc -Zlink=so -o sharedob function1.o function2.o \
    function3.o
cc file1.c file2.c file3.c /path/sharedob
```

we recommend against it. Not only will you have to enter a path name on the **`cc`** command line every time you use **`sharedob`** in a program, that path name will be hard-coded in your executables. The reason why you want to avoid this is related to our next point.

We said that the command line

```
cc -L dir file1.c file2.c file3.c -lfoo
```

would direct the link editor to record in your executable the name of the shared object with which it is to be linked at run time. Note: the *name* of the shared object, not its path name. What this means is that when you use the **-l** option to link your program with a shared object library, not only must the link editor be told which directory to search for that library, so must the dynamic linker (unless the directory is the standard place, which the dynamic linker searches by default). We'll show you how to point the dynamic linker to directories in the subsection "Specifying Directories to Be Searched by the Dynamic Linker" on page 4-15. What we want to stress here is that as long as the path name of a shared object is not hard-coded in your executable, you can move the shared object to a different directory without breaking your program. That's the main reason why you should avoid using path names of shared objects on the **cc** command line. Those path names will be hard-coded in your executable. They won't be if you use **-l**.

Finally, the **cc -Zlink=so** command will not only create a shared object, it will accept a shared object or archive library as input. In other words, when you create **libfoo.so**, you can link it with a library you have already created, say, **libsharedob.so**:

```
cc -Zlink=so -o libfoo.so -L dir function1.o function2.o\
    function3.o -lsharedob
```

That command will arrange for **libsharedob.so** to be linked with **libfoo.so** when, at run time, **libfoo.so** is linked with your program. Note that here you will have to point the dynamic linker to the directories in which both **libfoo.so** and **libsharedob.so** are stored.

Specifying Directories to Be Searched by the Link Editor

In the previous subsection we created the archive library **libfoo.a** and the shared object **libfoo.so**. For the sake of discussion, we'll now say that both these libraries are stored in the directory **/home/mylibs**. We'll also assume that you are creating your executable in a different directory. In fact, these assumptions are not academic. They reflect the way most programmers organize their work on the PowerUX system.

The first thing you must do if you want to link your program with either of these libraries is point the link editor to the **/home/mylibs** directory by specifying its path name with the **-L** option:

```
cc -L /home/mylibs file1.c file2.c file3.c -lfoo
```

The **-L** option directs the link editor to search for the libraries named with **-l** first in the specified directory, then in the standard places. In this case, having found the directory **/home/mylibs**, the link editor will search **libfoo.so** rather than **libfoo.a**. As we saw earlier, when the link editor encounters otherwise identically named shared object and archive libraries in the same directory, it searches the library with the **.so** suffix by default. For the same reason, it will search **libc.so** here rather than **libc.a**. Note that you must specify **-L** if you want the link editor to search for libraries in your current directory. You can use a period (**.**) to represent the current directory.

To direct the link editor to search **libfoo.a**, you can turn off the dynamic linking default:

```
cc -Zlink=static -L /home/mylibs file1.c file2.c \
    file3.c -lfoo
```

Under **-Zlink=static**, the link editor will not accept shared objects as input. It will search **libfoo.a** rather than **libfoo.so**, and **libc.a** rather than **libc.so**.

To link your program statically with **libfoo.a** and dynamically with **libc.so**, you can do either of two things. First, you can move **libfoo.a** to a different directory--**/home/archives**, for example--then specify **/home/archives** with the **-L** option:

```
cc -L /home/archives -L /home/mylibs file1.c file2.c \
    file3.c -lfoo
```

As long as the link editor encounters the **/home/archives** directory before it encounters the **/home/mylibs** directory, it will search **libfoo.a** rather than **libfoo.so**. That is, when otherwise identically named **.so** and **.a** libraries exist in your directories, the link editor will search the first one it finds. The same thing is true, by the way, for identically named libraries of either type. If you have different versions of **libfoo.a** in your directories, the link editor will search the first one it finds.

A better alternative might be to leave **libfoo.a** where you had it in the first place and use the **-Zlibs=static** and **-Zlibs=dynamic** options to turn dynamic linking off and on. The following command will link your program statically with **libfoo.a** and dynamically with **libc.so**:

```
cc -L /home/mylibs file1.c file2.c file3.c \
    -Zlibs=static -lfoo -Zlibs=dynamic
```

When you specify **-Qstatic**, the link editor will not accept a shared object as input until you specify **-Qdynamic**. In other words, you can use these options as toggles--any number of times--on the **cc** command line:

```
cc -L /home/mylibs file1.c file2.c -Zlibs=static -lfoo \
    file3.c -Zlibs=dynamic -lsharedob
```

That command will direct the link editor to search

- First, **libfoo.a** to resolve still unresolved external references in **file1.c** and **file2.c**;
- Second, **libsharedob.so** to resolve still unresolved external references in all three files and in **libfoo.a**;
- Last, **libc.so** to resolve still unresolved external references in all three files and the preceding libraries.

Files, including libraries, are searched for definitions in the order they are listed on the **cc** command line. The standard C library is always searched last.

You can add to the list of directories to be searched by the link editor by using the environment variable LD_LIBRARY_PATH. LD_LIBRARY_PATH must be a list of colon-separated directory names; an optional second list is separated from the first by a semicolon:

LD_LIBRARY_PATH=*dir***:***dir/***;***dir***:***dir***;export LD_LIBRARY_PATH**

The directories specified before the semicolon are searched, in order, before the directories specified with **-L**; the directories specified after the semicolon are searched, in order, after the directories specified with **-L**. Note that you can use LD_LIBRARY_PATH in place of **-L** altogether. In that case the link editor will search for libraries named with **-l** first in the directories specified before the semicolon, next in the directories specified after the semicolon, and last in the standard places. You should use absolute path names when you set this environment variable.

NOTE

As we explain in the next subsection, `LD_LIBRARY_PATH` is also used by the dynamic linker. That is, if `LD_LIBRARY_PATH` exists in your environment, the dynamic linker will search the directories named in it for shared objects to be linked with your program at execution. In using `LD_LIBRARY_PATH` with the link editor or the dynamic linker, then, you should keep in mind that any directories you give to one you are also giving to the other.

Specifying Directories to Be Searched by the Dynamic Linker

Earlier we said that when you use the **-l** option, you must point the dynamic linker to the directories of the shared objects that are to be linked with your program at execution. The environment variable `LD_RUN_PATH` lets you do that at link time. To set `LD_RUN_PATH`, list the absolute path names of the directories you want searched in the order you want them searched. Separate path names with a colon. Since we are concerned only with the directory **/home/mylibs** here, the following will do:

```
LD_RUN_PATH=/home/mylibs;export LD_RUN_PATH
```

Now the command

```
cc -o prog -L /home/mylibs file1.c file2.c file3.c -lfoo
```

will direct the dynamic linker to search for **libfoo.so** in **/home/mylibs** when you execute your program:

```
prog
```

The dynamic linker searches the standard place by default, after the directories you have assigned to `LD_RUN_PATH`. Note that as far as the dynamic linker is concerned, the standard place for libraries is **/usr/lib**. Any executable versions of libraries supplied by the compilation system are kept in **/usr/lib**.

The environment variable `LD_LIBRARY_PATH` lets you do the same thing at run time. Suppose you have moved **libfoo.so** to **/home/sharedobs**. It is too late to replace **/home/mylibs** with **/home/sharedobs** in `LD_RUN_PATH`, at least without link editing your program again. You can, however, assign the new directory to `LD_LIBRARY_PATH`, as follows:

```
LD_LIBRARY_PATH=/home/sharedobs;export LD_LIBRARY_PATH
```

Now when you execute your program

```
prog
```

the dynamic linker will search for **`libfoo.so`** first in **`/home/mylibs`** and, not finding it there, in **`/home/sharedobs`**. That is, the directory assigned to `LD_RUN_PATH` is searched before the directory assigned to `LD_LIBRARY_PATH`. The important point is that because the path name of **`libfoo.so`** is not hard-coded in **`prog`**, you can direct the dynamic linker to search a different directory when you execute your program. In other words, you can move a shared object without breaking your application.

You can set `LD_LIBRARY_PATH` without first having set `LD_RUN_PATH`. The main difference between them is that once you have used `LD_RUN_PATH` for an application, the dynamic linker will search the specified directories every time the application is executed (unless you have relinked the application in a different environment). In contrast, you can assign different directories to `LD_LIBRARY_PATH` each time you execute the application. `LD_LIBRARY_PATH` directs the dynamic linker to search the assigned directories before it searches the standard place. Directories, including those in the optional second list, are searched in the order listed. See the previous subsection for the syntax.

Note, finally, that when linking a set-user or set-group ID program, the dynamic linker will ignore any directories specified by `LD_LIBRARY_PATH` that are not “trusted.” Trusted directories are built into the dynamic linker and cannot be modified by the application. Currently, the only trusted directory is **`/usr/lib`**.

Checking for Run-Time Compatibility

Suppose you have been supplied with an updated version of a shared object. You have already compiled your program with the previous version; the link editor has checked it for undefined symbols, found none, and created an executable. According to everything we have said, you should not have to link your program again. The dynamic linker will simply use the definitions in the new version of the shared object to satisfy unresolved external references in the executable.

Suppose further that this is a database update program that takes several days to run. You want to be sure that your program does not fail in a critical section because a symbol that was defined by the previous version of the shared object is no longer defined by the new version. In other words, you want the information that the link editor gives you--that your executable is compatible with the shared library--without having to link edit it again.

There are two ways you can check for run-time compatibility. The command **`ldd(1)`** (“list dynamic dependencies”) directs the dynamic linker to print the path names of the shared objects on which your program depends:

```
ldd prog
```

When you specify the **`-d`** option to **`ldd(1)`**, the dynamic linker prints a diagnostic message for each unresolved data reference it would encounter if *prog* were executed. When you specify the **`-r`** option, it prints a diagnostic message for each unresolved data or function reference it would encounter if *prog* were executed. You can do the same thing when you execute your program. Whereas the dynamic linker resolves data references immediately at run time, it normally delays resolving function references until a function is invoked for the first time. Normally, then, the lack of a definition for a function will not be apparent until the function is invoked. By setting the environment variable `LD_BIND_NOW`

```
LD_BIND_NOW=1;export LD_BIND_NOW
```

before you execute your program, you direct the dynamic linker to resolve all references immediately. In that way, you can learn before execution of `main()` begins that the functions invoked by your process actually are defined.

Dynamic Linking Programming Interface

You can use a programming interface to the dynamic linking mechanism to attach a shared object to the address space of your process during execution, look up the address of a function in the library, call that function, and then detach the library when it is no longer needed. The routines for this are stored in **`libdl.so`**. Subsection 3X man pages describe its contents.

Implementation

We have already described, in various contexts in this section, the basic implementation of the static and dynamic linking mechanisms:

- When you use an archive library function, a copy of the object file that contains the function is incorporated in your executable at link time. External references to the function are assigned virtual addresses when the executable is created.
- When you use a shared library function, the entire contents of the library are mapped into the virtual address space of your process at run time. External references to the function are assigned virtual addresses when you execute the program. The link editor records in your executable only the name of the shared object and a small amount of bookkeeping information for use by the dynamic linker at run time.

We'll take a closer look at how dynamic linking is implemented in a moment. First let's consider the one or two cases in which you might not want to use it. Earlier we said that because shared object code is not copied into your executable object file at link time, a dynamically linked executable normally will use less disk space than a statically linked executable. If your program calls only a few small library functions, however, the bookkeeping information to be used by the dynamic linker may take up more space in your executable than the code for those functions. You can use the **`size(1)`** command to determine the difference.

In a similar way, using a shared object may occasionally add to the memory requirements of a process. Although a shared object's text is shared by all processes that use it, its data typically are not (at least its writable data; see the subsection “Guidelines for Building Shared Objects” on page 4-18 for the distinction). Every process that uses a shared object usually gets a private copy of its entire data segment, regardless of how many of the data are needed. If an application uses only a small portion of a shared library's text and data, executing the application might require more memory with a shared object than without one. It would be unwise, for example, to use the standard C shared object library to access only `strcmp()`. Although sharing `strcmp()` saves space on your disk and memory on the system, the memory cost to your process of having a private copy of the C library's data segment would make the archive version of `strcmp()` the more appropriate choice.

Now let's consider dynamic linking in a bit more detail. First, each process that uses a shared object references a single copy of its code in memory. That means that when other users on your system call a function in a shared object library, the entire contents of that library are mapped into the virtual address space of their processes as well. If they have called the same function as you, external references to the function in their programs will, in all likelihood, be assigned different virtual addresses. That is, because the function may be loaded at a different virtual address for each process that uses it, the system cannot calculate absolute addresses in memory until run time.

Second, the memory management scheme underlying dynamic linking shares memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at run time. If a process writes to a shared page in the course of relocating a reference to a shared object, it gets a private copy of that page and loses the benefits of code sharing (although without affecting other users of the page).

Third, to create programs that require the least possible amount of page modification at run time, the compiler generates position-independent code under the **`-ZPIC`** option. Whereas executable code normally must be tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to specific addresses, it will execute correctly--without page modification--at a different address in each process that uses it. As we have indicated, you should specify **`-ZPIC`** when you create a shared object:

```
cc -ZPIC -Zlink=so -o libfoo.so function1.c function2.c\
    function3.c
```

Relocatable references in your object code will be moved from its text segment to tables in the data segment. See Chapter 22 ("Executable and Linking Format (ELF)") in this manual for the details. In the next subsection we'll look at some basic guidelines for building shared objects. For now, we'll sum up the reasons why you might want to use one:

- Because library code is not copied into the executables that use it, they require less disk space.
- Because library code is shared at run time, the dynamic memory needs of systems are reduced.
- Because symbol resolution is put off until run time, shared objects can be updated without having to relink applications that depend on them.
- As long as its path name is not hard-coded in an executable, a shared object can be moved to a different directory without breaking an application.

Guidelines for Building Shared Objects

This subsection gives coding guidelines and maintenance tips for shared library development. Before getting down to specifics, we should emphasize that if you plan to develop a commercial shared library, you ought to consider providing a compatible archive as well. As we have noted, some users may not find a shared library appropriate for their applications. Others may want their applications to run on PowerUX system releases without shared object support. Shared object code is completely compatible with archive library code. In other words, you can use the same source files to build archive and shared object versions of a library.

Let's look at some performance issues first. There are two things you want to do to enhance shared library performance:

Minimize the Library's Data Segment

As noted, only a shared object's text segment is shared by all processes that use it; its data segment typically is not. Every process that uses a shared object usually gets a private memory copy of its entire data segment, regardless of how many of the data are needed. You can cut down the size of the data segment a number of ways:

- Try to use automatic (stack) variables. Don't use permanent storage if automatic variables will work.
- Use functional interfaces rather than global variables. Generally speaking, that will make library interfaces and code easier to maintain. Moreover, defining functional interfaces often eliminates global variables entirely, which in turn eliminates global "copy" data. The ANSI C function **`strerror(3C)`** illustrates these points.

In previous implementations, system error messages were made available to applications only through two global variables:

```
extern int   sys_nerr;
extern char  *sys_errlist[];
```

That is, `sys_errlist[X]` gives a character string for the error `X`, if `X` is a non-negative value less than `sys_nerr`. Now if the current list of messages were made available to applications only through a lookup table in an archive library, applications that used the table obviously would not be able to access new messages as they were added to the system unless they were relinked with the library. In other words, errors might occur for which these applications could not produce meaningful diagnostics. Something similar happens when you use a global lookup table in a shared library.

First, the compilation system sets aside memory for the table in the address space of each executable that uses it, even though it does not know yet where the table will be loaded. After the table is loaded, the dynamic linker copies it into the space that has been set aside. Each process that uses the table, then, gets a private copy of the library's data segment, including the table, and an additional copy of the table in its own data segment. Moreover, each process pays a performance penalty for the overhead of copying the table at run time. Finally, because the space for the table is allocated when the executable is built, the application will not have enough room to hold any new messages you might want to add in the future. A functional interface overcomes these difficulties. `strerror()` might be implemented as follows:

```
static const char *msg[] = {
    "Error 0",
    "Not owner",
    "No such file or directory",
    ...
};
```

```
char * strerror(int err)
{
    if (err < 0 || err >= sizeof(msg)/sizeof(msg[0]))
        return 0;
    return (char *)msg[err];
}
```

The message array is static, so no application space is allocated to hold a separate copy. Because no application copy exists, the dynamic linker does not waste time moving the table. New messages can be added, because only the library knows how many messages exist. Finally, note the use of the type qualifier `const` to identify data as read-only. Whereas writable data are stored in a shared object's data segment, read-only data are stored in its text segment. For more on `const`, see the Concurrent *C Reference Manual*.

In a similar way, you should try to allocate buffers dynamically--at run time--instead of defining them at link time. That will save memory because only the processes that need the buffers will get them. It will also allow the size of the buffers to change from one release of the library to the next without affecting compatibility. Example:

```
char * buffer()
{
    static char *buf = 0;

    if (buf = = 0)
    {
        if ((buf = malloc(BUFSIZE)) = = 0)
            return 0;
    }
    ...
    return buf;
}
```

Exclude functions that use large amounts of global data--that is, if you cannot rewrite them in the ways described in the foregoing items. If an infrequently used routine defines a great deal of static data, it probably does not belong in a shared library.

Make the library self-contained. If a shared object imports definitions from another shared object, each process that uses it will get a private copy not only of its data segment, but of the data segment of the shared object from which the definitions were imported. In cases of conflict, this guideline should probably take precedence over the preceding one.

Minimize Paging Activity

Although processes that use shared libraries will not write to shared pages, they still may incur page faults. To the extent they do, their performance will degrade. You can minimize paging activity in the following ways:

- Organize to improve locality of reference. First, exclude infrequently used routines on which the library itself does not depend. Traditional **`a.out`** files contain all the code they need at run time. So if a process calls a function, it may already be in memory because of its proximity to other text in the process. If the function is in a shared library, however, the surrounding library code may be unrelated to the calling process. Only rarely, for example, will any single executable use everything in the shared C library. If a shared library has unrelated functions, and if unrelated processes make random calls to those functions, locality of reference may be decreased, leading to more paging activity. The point is that functions used by only a few **`a.out`** files do not save much disk space by being in a shared library, and can degrade performance.

Second, try to improve locality of reference by grouping dynamically related functions. If every call to `funcA()` generates calls to `funcB()` and `funcC()`, try to put them in the same page. **`cflow(1)`** generates this kind of static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

- Align for paging. Try to arrange the shared library's object files so that frequently used functions do not unnecessarily cross page boundaries. First, determine where the page boundaries fall. The page size is 4K. You can use the **`nm(1)`** command to determine how symbol values relate to page boundaries. After grouping related functions, break them up into page-sized chunks. Although some object files and functions are larger than a page, many are not. Then use the less frequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault is decreased. You can put frequently used, unrelated functions together because they will probably be called randomly enough to keep the pages in memory.
- Avoid hardware thrashing. You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segments mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry again.

Now let's look at some maintenance issues. We have already seen how allocating buffers dynamically can ease the job of library maintenance. As a general rule, you want to be sure that updated versions of a shared object are compatible with its previous versions so that users will not have to recompile their applications. At the very least, you should avoid changing the names of library symbols from one release to the next. All the same, there may be instances in which you need to release a library version that is incompatible with its predecessor. On the one hand, you will want to maintain the older version for dynamically linked executables that depend on it. On the other hand, you will want newly created executables to be linked with the updated version. Moreover, you will probably want both versions to be stored in the same directory. In this situation, you could give the new release a different name, rewrite your documentation, and so forth. A better alternative would be to plan for the contingency in the very first instance by using the following sequence of commands when you create the original version of the shared object:

```
cc -ZPIC -Zlink=so -h libfoo.1 -o libfoo.1 function1.c \
    function2.c function3.c
ln libfoo.1 libfoo.so
```

In the first command **`-h`** stores the name given to it, **`libfoo.1`**, in the shared object itself. You then use the UNIX system command **`ln(1)`** to create a link between the name

libfoo.1 and the name **libfoo.so**. The latter, of course, is the name the link editor will look for when users of your library specify

```
cc -L dir file1.c file2.c file3.c -lfoo
```

In this case, however, the link editor will record in the user's executable the name you gave to **-h**, **libfoo.1**, rather than the name **libfoo.so**. That means that when you release a subsequent, incompatible version of the library, **libfoo.2**, executables that depend on **libfoo.1** will continue to be linked with it at run time. As we saw earlier, the dynamic linker uses the shared object name that is stored in the executable to satisfy unresolved external references at run time.

You use the same sequence of commands when you create **libfoo.2**:

```
cc -ZPIC -Zlink=so -h libfoo.2 -o libfoo.2 function1.c \
    function2.c function4.c
ln libfoo.2 libfoo.so
```

Now when users specify

```
cc -L dir file1.c file2.c file3.c -lfoo
```

The name **libfoo.2** will be stored in their executables, and their programs will be linked with the new library version at run time.

Multiply-Defined Symbols

Multiply-defined symbols--except for different-sized initialized data objects--are not reported as errors under dynamic linking. To put that more formally, the link editor will not report an error for multiple definitions of a function or a same-sized data object when each such definition resides within a different shared object or within a dynamically linked executable and different shared objects. The dynamic linker will use the definition in whichever object occurs first on the **cc** command line. You can, however, specify **-Qsymbolic** when you create a shared object

```
cc -ZPIC -Zlink=so -Zsymbolic -o libfoo.so function1.c \
    function2.c function3.c
```

to insure that the dynamic linker will use the shared object's definition of one of its own symbols, rather than a definition of the same symbol in an executable or another library.

In contrast, multiply-defined symbols are generally reported as errors under static linking. We say "generally" because definitions of so-called weak symbols can be hidden from the link editor by a definition of a global symbol. That is, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error.

To illustrate this, let's look at our own implementation of the standard C library. This library provides services that users are allowed to redefine and replace. At the same time, however, ANSI C defines standard services that must be present on the system and cannot be replaced in a strictly conforming program. `fread()`, for example, is an ANSI C library function; the system function `read()` is not. So a conforming program may redefine `read()` and still use `fread()` in a predictable way.

The problem with this is that `read()` underlies the `fread()` implementation in the standard C library. A program that redefines `read()` could "confuse" the `fread()` implementation. To guard against this, ANSI C states that an implementation cannot use a name that is not reserved to it. That's why we use `_read()`--note the leading underscore--to implement `fread()` in the standard C library.

Now suppose that a program you have written calls `read()`. If your program is going to work, a definition for `read()` does exist in the C library. It is identical to the definition for `_read()` and contained in the same object file.

Suppose further that another program you have written redefines `read()`, as it has every right to do under ANSI C; this same program calls `fread()`. Because you get our definitions of both `_read()` and `read()` when you use `fread()`, we would expect the link editor to report the multiply-defined symbol `read()` as an error, and fail to create an executable program. To prevent that, we used the `#pragma` directive in our source code for the library as follows:

```
#pragma weak read = _read
```

Because our `read()` is defined as a weak symbol, your own definition of `read()` will override the definition in the standard C library. You can use the `#pragma` directive in the same way in your own library code.

There's a second use for weak symbols that you ought to know about:

```
#pragma weak read
```

tells the link editor not to complain if it does not find a definition for the weak symbol read. References to the symbol use the symbol value if defined, `0` otherwise. The link editor does not extract archive members to resolve undefined weak symbols. The mechanism is intended to be used primarily with functions. Although it will work for most data objects, it should not be used with uninitialized global data ("common" symbols) or with shared library data objects that are exported to executables.

Mapfiles

The link editor (**`ld`**) automatically and intelligently maps input sections from object files (**`.o`** files) to output segments in executable files (**`a.out`** files). The **`mapfile`** option to the **`ld`** command allows you to change the default mapping provided by the link editor.

In particular, the **`mapfile`** option allows you to:

- Declare segments and specify values for segment attributes such as segment type, permissions, addresses, length, and alignment
- Control mapping of input sections to segments by specifying the attribute values necessary in a section to map to a specific segment (the attributes are section name, section type, and permissions) and by specifying which object file(s) the input sections should be taken from, if necessary
- Declare a global-absolute symbol that is assigned a value equal to the size of a specified segment (by the link editor) and that can be referenced from object files

NOTE

The major purpose of the **`mapfile`** option is to allow users of **`ifiles`** (an option previously available to **`ld`** that used link editor command language directives) to convert to mapfiles. All other facilities previously available for **`ifiles`**, other than those mentioned above, are not available with the **`mapfile`** option.

When using the **`mapfile`** option, be aware that you can easily create **`a.out`** files that do not execute. Therefore, the use of the **`mapfile`** option is strongly discouraged. **`ld`** knows how to produce a correct **`a.out`** without the use of the **`mapfile`** option. The **`mapfile`** option is intended for system programming use, not application programming use.

This subsection describes the structure and syntax of a mapfile and the use of the **`-M`** option to the **`ld`** command.

Using the Mapfile Option

To use the **`mapfile`** option, you must:

1. Enter mapfile directives into a file (this is your “mapfile”)
2. Enter the following option on the **`ld`** command line:

   `-M` *mapfile*

   mapfile is the file name of the file you produced in step 1. If the *mapfile* is not in your current directory, you must include the full path name; no default search path exists. (See the **`ld(1)`** for information on operation of the **`ld`** command.)

Mapfile Structure and Syntax

You can enter three types of directives into a mapfile:

- Segment declarations
- Mapping directives
- Size-symbol declarations

Each directive can span more than one line and can have any amount of white space (including new-lines) as long as it is followed by a semicolon. You can enter `0` (zero) or more directives in a mapfile. (Entering 0 directives causes **`ld`** to ignore the mapfile and use its own defaults.) Typically, segment declarations are followed by mapping directives, i.e., you would declare a segment and then define the criteria by which a section becomes part of that segment. If you enter a mapping directive or size-symbol declaration without first declaring the segment to which you are mapping (except for built-in segments, explained later), the segment is given default attributes as explained below. This segment is then an *implicitly declared segment*.

Size-symbol declarations can appear anywhere in a mapfile.

The following sections describe each directive type. For all syntax discussions, the following apply:

- All entries in "constant width", all colons, semicolons, equal signs, and at (`@`) signs are typed in literally.
- All entries in italics are "substitutables."
- `{ ... }*` means "zero or more."
- `{ ... }+` means "one or more."
- `[ ... ]` means "optional."
- *section_names* and *segment_names* follow the same rules as C identifiers where a period (`.`) is treated as a letter (e.g., `.bss` is a legal name).
- *section_names*, *segment_names*, *file_names*, and *symbol_names* are case sensitive; everything else is not case sensitive.
- Spaces (or new-lines) may appear anywhere except before a number or in the middle of a name or value.
- Comments beginning with # and ending at a new-line may appear anywhere that a space may appear.

Segment Declarations

A segment declaration creates a new segment in the **`a.out`** or changes the attribute values of an existing segment. (An existing segment is one that you previously defined or one of the three built-in segments described below.)

A segment declaration has the following syntax:

segment_name `=` `{`*segment_attribute_value*`}*;`

For each *segment_name*, you can specify any number of *segment_attribute_values* in any order, each separated by a space. (Only one attribute value is allowed for each segment attribute.) The segment attributes and their valid values are as follows:

"*segment_type*`:`"	
	`LOAD`
	`NOTE`
"*segment_flags*`:`"	
	`?[R][W][X]`
"*virtual_address*`:`"	
	`V`*number*
"*physical_address*`:`"	
	`P`*number*
"*length*`:`"	
	`L`*number*
"*alignments*`:`"	
	`A`*number*

There are three built-in segments with the following default attribute values:

- `text` (`LOAD`, `?RX`, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- `data` (`LOAD`, `?RWX`, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- `note` (`NOTE`)

`ld` behaves as if these segments had been declared before your mapfile is read in. See "Mapfile Option Defaults" on page 4-30 for more information.

Note the following when entering segment declarations:

- A *number* can be hexadecimal, decimal, or octal, following the same rules as in the C language.
- No space is allowed between the `V`, `P`, `L`, or `A` and the *number.*
- The *segment_type* value can be either `LOAD` or `NOTE`.
- The *segment_type* value defaults to `LOAD`.
- The *segment_flags* values are `R` for readable, `W` for writable, and `X` for executable. No spaces are allowed between the question mark and the individual flags that make up the *segment_flags* value.
- The *segment_flags* value for a `LOAD` segment defaults to `RWX`.
- `NOTE` segments cannot by assigned any segment attribute value other than a *segment_type*.
- Implicitly declared segments default to *segment_type* value `LOAD`, *segment_flags* value `RWX`, a default *virtual_address*, *physical_address*, and *alignment* value, and have no *length* limit.

  `ld` calculates the addresses and length of the current segment based on the previous segment's attribute values. Also, even though implicitly declared segments default to "no length limit," any machine memory limitations still apply.
- `LOAD` segments can have an explicitly specified *virtual_address* value and/or *physical_address* value, as well as a maximum segment *length* value.
- If a segment has a *segment_flags* value of `?` with nothing following, the value defaults to not readable, not writable and not executable.
- The *alignment* value is used in calculating the virtual address of the beginning of the segment. This alignment only affects the segment for which it is specified; other segments still have the default alignment unless their alignments are also changed.
- If any of the *virtual_address*, *physical_address*, or *length* attribute values are not set, **`ld`** calculates these values as it builds the **`a.out`**.
- If an *alignment* value is not specified for a segment, it is set to the built-in default. (The default differs from one CPU to another and may even differ between kernel versions. You should check the appropriate documentation for these numbers).

- If both a *virtual_address* and an *alignment* value are specified for a segment, the *virtual_address* value takes priority.
- If a *virtual_address* value is specified for a segment, the alignment field in the program header contains the default *alignment* value.

NOTE

If a *virtual_address* value is specified, the segment is placed at that virtual address. For the PowerUX system kernel, this creates a correct result. For files that start via `exec()`, this method creates an incorrect **`a.out`** file because the segments do not have correct offsets relative to their page boundaries.

Mapping Directives

A mapping directive tells **`ld`** how to map input sections to segments. Basically, you name the segment that you are mapping to and indicate what the attributes of a section must be in order to map into the named segment. The set of *section_attribute_values* that a section must have to map into a specific segment is called the entrance criteria for that segment. In order to be placed in a specified segment of the **`a.out`**, a section must meet the entrance criteria for a segment exactly.

A mapping directive has the following syntax:

segment_name `:` `{`*section_attribute_value*`}*` `[:` `{`*file_name*`}+];`

For a *segment_name*, you specify any number of *section_attribute_values* in any order, each separated by a space. (At most one section attribute value is allowed for each section attribute.) You can also specify that the section must come from a certain **`.o`** file(s) via the *file_name* substitutable. The section attributes and their valid values are as follows:

"*section_name*`:`"	
	any valid section name
"*section_type*`:`"	
	`$PROGBITS`
	`$SYMTAB`
	`$STRTAB`
	`$REL`
	`$RELA`
	`$NOTE`
	`$NOBITS`
"*section_flags*`:`"	
	`?[[!]A][[!]W][[!]X]`

Note the following when entering mapping directives:

- You must choose at most one *section_type* from the *section_types* listed above. The *section_types* listed above are built-in types. For more information on *section_types*, see Chapter 22 ("Executable and Linking Format (ELF)").
- The *section_flags* values are `A` for allocatable, `W` for writable, or `X` for executable. If an individual flag is preceded by an exclamation mark (`!`),

the link editor checks to make sure that the flag is not set. No spaces are allowed between the question mark, exclamation point(s), and the individual flags that make up the *section_flags* value.

- *file_name* may be any legal file name and can be of the form *archive_name(component_name)*, e.g., **`/lib/libc.a`**(**`printf.o`**). A file name may be of the form **file_name* (see next bullet item). Note that **`ld`** does not check the syntax of file names.
- If a *file_name* is of the form **file_name*, **`ld`** simulates a basename (see **`basename(1)`**) on the file name from the command line and uses that to match against the mapfile *file_name*. In other words, the *file_name* from the mapfile only needs to match the last part of the file name from the command line. (See "Mapping Example" on page 4-29.)
- If you use the **`-l`** option on the **`cc`** or **`ld`** command line, and the library after the **`-l`** option is in the current directory, you must precede the library with **`./`** (or the entire path name) in the mapfile in order to create a match.
- More than one directive line may appear for a particular output segment, e.g., the following set of directives is legal:

  ```
  S1 : $PROGBITS;
  S1 : $NOBITS;
  ```

  Entering more than one mapping directive line for a segment is the only way to specify multiple values of a section attribute.
- A section can match more than one entrance criteria. In this case, the first segment encountered in the mapfile with that entrance criteria is used, e.g., if a mapfile reads:

  ```
  S1 : $PROGBITS;
  S2 : $PROGBITS;
  ```

  the `$PROGBITS` sections are mapped to segment `S1`.

Extended Mapping Directives

PowerUX mapfiles support an extension to the set of mapping directives described above. These extensions permit the definition or redefinition of variables within a section. These extended directives are output by the **`shmdefine(1)`** utility.

Size-Symbol Declarations

Size-symbol declarations let you define a new global-absolute symbol that represents the size, in bytes, of the specified segment. This symbol can be referenced in your object files. A size-symbol declaration has the following syntax:

segment_name `@` *symbol_name* *symbol_name*

can be any legal C identifier, although the **`ld`** command does not check the syntax of the *symbol_name*.

Mapping Example

Figure 4-1 is an example of a user-defined mapfile. The numbers on the left are included in the example for tutorial purposes. Only the information to the right of the numbers would actually appear in the mapfile.

```
1.   elephant : .bss : peanuts.o *popcorn.o;

2.   monkey : $PROGBITS ?AX;
3.   monkey : .bss;
4.   monkey = LOAD V0x80000000 L0x40000;

5.   donkey : .bss;
6.   donkey = ?RX A0x1000;

7.   text = V0x80008000;
```

Figure 4-1. User-Defined Mapfile

Four separate segments are manipulated in this example. The implicitly declared segment `elephant` (line 1) receives all of the **`.bss`** sections from the files **`peanuts.o`** and **`popcorn.o`**. Note that **`*popcorn.o`** matches any **`popcorn.o`** file that may have been entered on the **`ld`** command line; the file need not be in the current directory. On the other hand, if **`/var/tmp/peanuts.o`** were entered on the **`ld`** command line, it would not match **`peanuts.o`** because it is not preceded by a `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections (not already in the segment `elephant`) with the name `.bss` (line 3). The **`.bss`** sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the *section_type* and *section_flags* values were entered on a separate line from the *section_name* value. (An *and* relationship exists between attributes on the same line as illustrated by `$PROGBITS` and `?AX` on line 2. An *or* relationship exists between attributes for the same segment that span more than one line as illustrated by `$PROGBITS ?AX` on line 2 or **`.bss`** on line 3.) The `monkey` segment is implicitly declared in line 2 with *segment_type* value `LOAD`, *segment_flags* value `RWX`, and no *virtual_address*, *physical_address*, *length* or *alignment* values specified (defaults are used). In line 4 the *segment_type* value of `monkey` is set to `LOAD` (since the *segment_type* attribute value does not change, no warning is issued), *virtual_address* value to `0x80000000` and maximum *length* value to `0x4000` (since the *length* attribute value changed, a warning is issued).

Line 5 implicitly declares the `donkey` segment. The entrance criteria is designed to route all **`.bss`** sections to this segment. Actually, no sections fall into this segment because the entrance criteria for `monkey` in line 3 capture all of these sections. In line 6, the *segment_flags* value is set to `?RX` and the *alignment* value is set to `0x1000` (since both of these attribute values changed, a warning is issued).

Line 7 sets the *virtual_address* value of the `text` segment to `0x80008000` (no warning is issued here).

The example user-defined mapfile in Figure 4-1 is designed to cause warnings for illustration purposes. If you wanted to change the order of the directives to avoid warnings, the example would appear as follows:

```
1.      elephant : .bss : peanuts.o *popcorn.o;
4.      monkey = LOAD V0x80000000 L0x4000;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .bss;
6.      donkey = ?RX A0x1000;
5.      donkey : .bss;
7.      text = V0x80008000;
```

This order eliminates all warnings.

Mapfile Option Defaults

The **ld** command has three built-in segments (`text`, `data`, and `note`) with default *segment_attribute_values* and corresponding default mapping directives as described under "Segment Declarations" on page 4-25. Even though the **ld** command does not use an actual "mapfile" to store the defaults, the model of a "default mapfile" helps to illustrate what happens when the **ld** command encounters your mapfile.

Figure 4-2 shows how a mapfile would appear for the **ld** command defaults. The **ld** command begins execution behaving as if the mapfile in Figure 4-2 has already been read in. Then **ld** reads your mapfile and either augments or makes changes to the defaults.

NOTE

The `interp` segment, which precedes all others, and the `dynamic` segment, which follows the `data` segment, are not shown in Figure 4-2 and Figure 4-3 because you cannot manipulate them.

```
text = LOAD ?RX
text : $PROGBITS ?A!W

data = LOAD ?RW
data : $PROGBITS ?AW
data : $NOBITS ?AW

note = NOTE
note : $NOTE
```

Figure 4-2. Default Mapfile

As each segment declaration in your mapfile is read in, it is compared to the existing list of segment declarations as follows:

1. If the segment does not already exist in the mapfile, but another with the same *segment_type* value exists, the segment is added before all of the existing segments of the same *segment_type*.

2. If none of the segments in the existing mapfile has the same *segment_type* value as the segment just read in, then the segment is added by *segment_type* value to maintain the following order:

   ```
   1. INTERP
   2. LOAD
   3. DYNAMIC
   4. NOTE
   ```

3. If the segment is of *segment_type* `LOAD` and you have defined a *virtual_address* value for this `LOAD`able segment, the segment is placed before any `LOAD`able segments without a defined *virtual_address* value or with a higher *virtual_address* value, but after any segments with a *virtual_address* value that is lower.

As each mapping directive in your mapfile is read in, the directive is added after any other mapping directives that you already specified for the same segment but before the default mapping directives for that segment.

Internal Map Structure

One of the most important data structures in **`ld`** is the map structure. A default map structure, corresponding to the model default mapfile mentioned above, is used by **`ld`** when the command is executed. Then, if the mapfile option is used, **`ld`** parses the mapfile to augment and/or override certain values in the default map structure.

A typical (although somewhat simplified) map structure is illustrated in Figure 4-3. The “Entrance Criteria” boxes correspond to the information in the default mapping directives and the “Segment Attribute Descriptors” boxes correspond to the information in the default segment declarations. The “Output Section Descriptors” boxes give the detailed attributes of the sections that fall under each segment. The sections themselves are in circles.

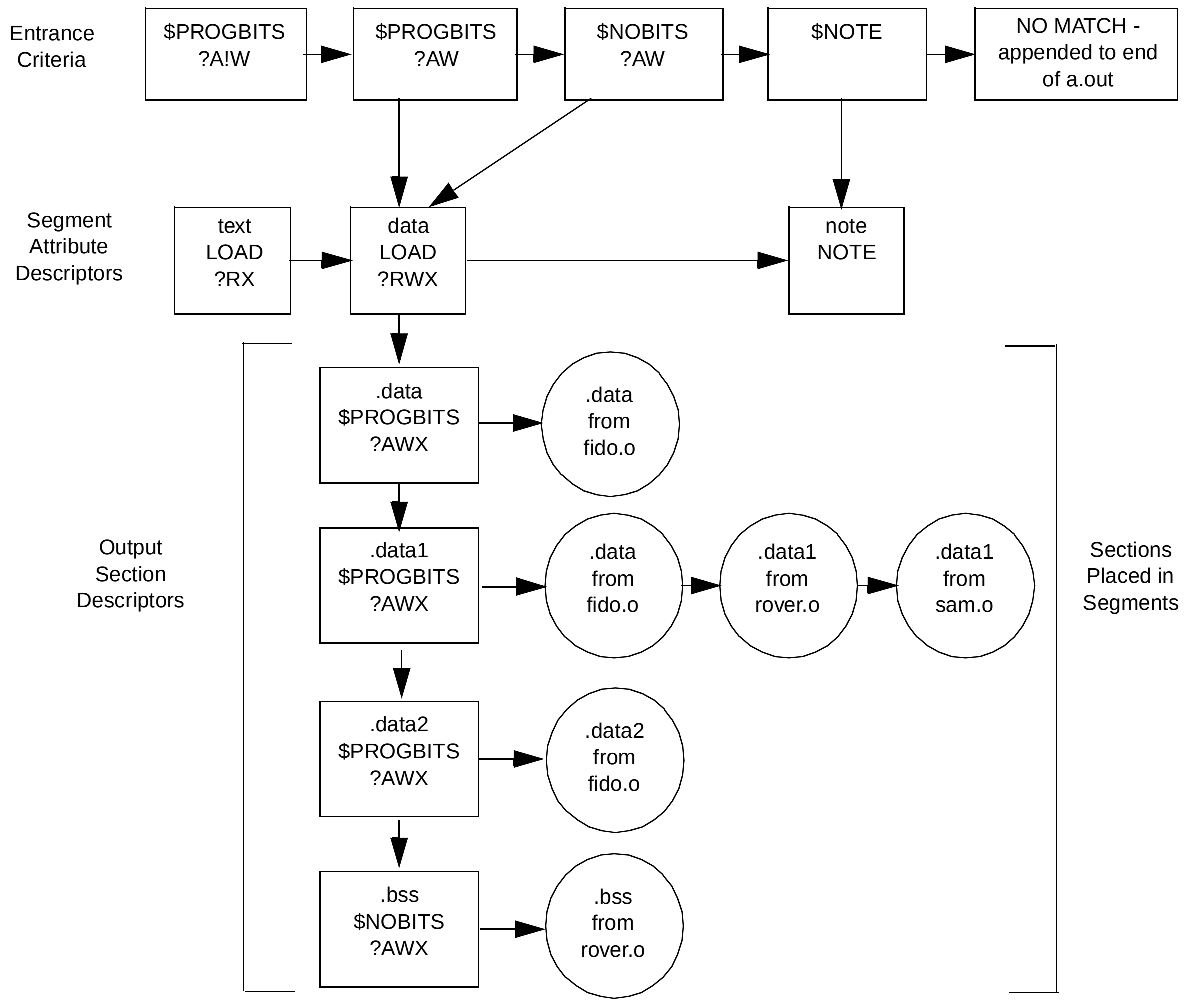


Figure 4-3. Simple Map Structure

`ld` performs the following steps when mapping sections to segments:

1. When a section is read in, **`ld`** checks the list of Entrance Criteria looking for a match. (All specified criteria must match):

   - In Figure 4-3, for a section to fall into the `text` segment it must have a *section_type* value of `$PROGBITS` and have a *section_flags* value of `?A!W`. It need not have the name **`.text`** since no name is specified in the Entrance Criteria. The section may be either `X` or `!X` (in the *section_flags* value) since nothing was specified for the execute bit in the Entrance Criteria.

   - If no Entrance Criteria match is found, the section is placed at the end of the `a.out` file after all other segments. (No program header entry is created for this information. See Chapter 22 (“Executable and Linking Format (ELF)”) for information on program headers.)

2. When the section falls into a segment, **`ld`** checks the list of existing Output Section Descriptors in that segment as follows:

   - If the section attribute values match those of an existing Output Section Descriptor exactly, the section is placed at the end of the list of sections associated with that Output Section Descriptor.

   - For instance, a section with a *section_name* value of `.data1`, a *section_type* value of `$PROGBITS`, and a *section_flags* value of `?AWX` falls into the second Entrance Criteria box in Figure 4-3, placing it in the `data` segment. The section matches the second Output Section Descriptor box exactly (`.data1`, `$PROGBITS`, `?AWX`) and is added to the end of the list associated with that box. The **`.data1`** sections from **`fido.o, rover.o`**, and **`sam.o`** illustrate this point.

   - If no matching Output Section Descriptor is found, but other Output Section Descriptors of the same *section_type* exist, a new Output Section Descriptor is created with the same attribute values as the section and that section is associated with the new Output Section Descriptor. The Output Section Descriptor (and the section) are placed after the last Output Section Descriptor of the same *section_type*. The **`.data2`** section in Figure 4-3 was placed in this manner.

   - If no other Output Section Descriptors of the indicated *section_type* exist, a new Output Section Descriptor is created and the section is placed so as to maintain the following *section_type* order:

     ```
     $DYNAMIC
     $PROGBITS
     $SYMTAB
     $STRTAB
     $RELA
     $REL
     $HASH
     $NOTE
     $NOBITS
     ```

     The **`.bss`** section in Figure 4-3 illustrates this point.

   NOTE

   If the input section has a user-defined *section_type* value (i.e., between `SHT_LOUSER` and `SHT_HIUSER`, see Chapter 22 ("Executable and Linking Format (ELF)")) it is treated as a `$PROGBITS` section. Note that no method exists for naming this *section_type* value in the mapfile, but these sections can be redirected using the other attribute value specifications (*section_flags*, *section_name*) in the entrance criteria.

3. If a segment contains no sections after all of the command line object files and libraries have been read in, no program header entry is produced for that segment.

NOTE

Input sections of type `$SYMTAB, $STRTAB, $REL`, and `$RELA` are used internally by **`ld`**. Directives that refer to these *section_types* can only map output sections produced by **`ld`** to segments.

Error Messages

When using the mapfile option, **`ld`** can return the following types of error messages:

Warnings	Do not stop execution of the link editor nor prevent the link editor from producing a viable **`a.out`**.
Fatal Errors	Stop execution of the link editor at the point at which the fatal error occurred.

Either `warning:` or `fatal:` appears at the beginning of each error message. Error messages are not numbered. The following conditions produce warnings:

- A *physical_address* or a *virtual_address* value or a *length* value appears for any segment other than a `LOAD` segment (the directive is ignored)
- A second declaration line exists for the same segment that changes an attribute value(s) (the second declaration overrides the original)
- An attribute value(s) (*segment_type* and/or *segment_flags* for `text` and `data`; *segment_type* for `note`) was changed for one of the built-in segments
- An attribute value(s) (*segment_type*, *segment_flags*, *length* and/or *alignment*) was changed for a segment created by an implicit declaration

The following conditions produce fatal errors:

- Specifying more than one **`-M`** option on the command line
- Specifying both the **`-r`** and the **`-M`** option on the same command line
- A mapfile cannot be opened or read
- A syntax error is found in the mapfile

NOTE

`ld` does not return an error if a *file_name*, *section_name*, *segment_name* or *symbol_name* does not conform to the rules in “Mapfile Structure and Syntax” on page 4-24 unless this condition produces a syntax error. For instance, if a name begins with a special character and this name is at the beginning of a directive line, **`ld`** returns an error. If the name is a *section_name* (appearing within the directive) **`ld`** does not return an error.

- More than one *segment_type*, *segment_flags*, *virtual_address*, *physical_address*, *length*, or *alignment* value appears on a single declaration line
- You attempt to manipulate either the `interp` segment or `dynamic` segment in a mapfile

NOTE

> The `interp` and `dynamic` segments are special built-in segments that you cannot change in any way.

- A segment grows larger than the size specified by your *length* attribute value
- A user-defined *virtual_address* value causes a segment to overlap the previous segment
- More than one *section_name*, *section_type*, or *section_flags* value appears on a single directive line
- A flag and its complement (e.g., `A` and `!A`) appear on a single directive line

Quick-Reference Guide

1. By convention, shared objects, or dynamically linked libraries, are designated by the prefix **`lib`** and the suffix **`.so`**; archives, or statically linked libraries, are designated by the prefix **`lib`** and the suffix **`.a.`** **`libc.so`**, then, is the shared object version of the standard C library; **`libc.a`** is the archive version.
2. These conventions are recognized, in turn, by the **`-l`** option to the **`cc`** command. That is, **`-l`***x* directs the link editor to search the shared object **`lib`***x***`.so`** or the archive library **`lib`***x***`.a`**. The **`cc`** command automatically passes **`-lc`** to the link editor. In other words, the compilation system arranges for the standard C library to be linked with your program transparently.
3. By default, the link editor chooses the shared object implementation of a library, **`lib`***x***`.so`**, in preference to the archive library implementation, **`lib`***x***`.a`**, in the same directory.
4. By default, the link editor searches for libraries in the standard places on your system, **`/usr/lib`** and **`/lib`**, in that order.

In this arrangement, then, C programs are dynamically linked with **`libc.so`** automatically:

```
cc file1.c file2.c file3.c
```

To link your program statically with **`libc.a`**, turn off the dynamic linking default with the **`-Zlink=static`** option:

```
cc -Zlink=static file1.c file2.c file3.c
```

Specify the **-l** option explicitly to link your program with any other library. If the library is in the standard place, the command

```
cc file1.c file2.c file3.c -lx
```

will direct the link editor to search for **lib***x***.so**, then **lib***x***.a** in the standard place. Note that the compilation system supplies shared object versions only of **libc** and **libdl**. (Other shared object libraries are supplied with the operating system, and usually are kept in the standard places.) Note too that as a rule it's best to place **-l** at the end of the command line.

If the library is not in the standard place, specify the path of the directory in which it is stored with the **-L** option

```
cc -L dir file1.c file2.c file3.c -lx
```

or the environment variable LD_LIBRARY_PATH

```
LD_LIBRARY_PATH=dir;export LD_LIBRARY_PATH
cc file1.c file2.c file3.c -lx
```

If the library is a shared object and is not in the standard place, you must also specify the path of the directory in which it is stored with either the environment variable LD_RUN_PATH at link time, or the environment variable LD_LIBRARY_PATH at run time:

```
LD_RUN_PATH=dir;export LD_RUN_PATH
LD_LIBRARY_PATH=dir;export LD_LIBRARY_PATH
```

It's best to use an absolute path when you set these environment variables. Note that LD_LIBRARY_PATH is read both at link time and at run time.

To direct the link editor to search **lib***x***.a** where **lib***x***.so** exists in the same directory, turn off the dynamic linking default with the **-Zlink=static** option:

```
cc -Zlink=static -L dir file1.c file2.c file3.c -lx
```

That command will direct the link editor to search **libc.a** as well as **lib***x***.a**. To link your program statically with **lib***x***.a** and dynamically with **libc.so**, use the **-Zlibs=static** and **-Zlibs=dynamic** options to turn dynamic linking off and on:

```
cc -L dir file1.c file2.c file3.c -Zlibs=static -lx \
    -Zlibs=dynamic
```

Files, including libraries, are searched for definitions in the order they are listed on the **cc** command line. The standard C library is always.

5
m4 Macro Processor

5
m4 Macro Processor

Introduction

m4 is a general purpose macro processor that can be used to preprocess C and assembly language programs, among other things. Besides the straightforward replacement of one string of text by another, **m4** lets you perform

- Integer arithmetic
- File inclusion
- Conditional macro expansion
- String and substring manipulation

You can use built-in macros to perform these tasks or define your own macros. Built-in and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process. A list of built-in macros appears on the **m4(1)** page.

The basic operation of **m4** is to read every legal token (string of ASCII letters and digits and possibly supplementary characters) and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

Macro calls have the general form

name`(`*arg1*`,` *arg2*`, ...,` *argn*`)`

If a macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that appear in the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

You invoke **m4** with a command of the form

m4 *file file file*

Each argument file is processed in order. If there are no arguments or if an argument is a hyphen, the standard input is read. If you are eventually going to compile the **m4** output, you could use a command something like this:

```
m4 file1.m4 > file1.c
```

You can use the **-D** option to define a macro on the **m4** command line. Suppose you have two similar versions of a program. You might have a single **m4** input file capable of generating the two output files. For example, **file1.m4** could contain lines such as

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

(**makefiles** are discussed in Chapter 13 (“Managing File Interactions with make”).) Your **makefile** for the program might look like this:

```
file1.1.c : file1.m4
            m4 -DVER=1 file1.m4 > file1.1.c
            ...

file1.2.c : file1.m4
            m4 -DVER=2 file1.m4 > file1.2.c
            ...
```

You can use the **-U** option to “undefine” VER. If **file1.m4** contains

```
if(VER, 1, do_something)
if(VER, 2, do_something)
ifndef(VER, do_something)
```

then your **makefile** would contain

```
file0.0.c : file1.m4
            m4 -UVER file1.m4 > file1.0.c
            ...

file1.1.c : file1.m4
            m4 -DVER=1 file1.m4 > file1.1.c
            ...

file1.2.c : file1.m4
            m4 -DVER=2 file1.m4 > file1.2.c
            ...
```

m4 Macros

Defining Macros

The primary built-in **m4** macro is `define()`, which is used to define new macros. The following input `define(`*name, stuff*`)`

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. The defined string must contain only ASCII alphanumeric or printable supplementary characters and must begin with a letter or printable supplementary character (underscore counts as a letter). The defining string is any text that contains balanced parentheses; it may stretch over multiple lines. As a typical example

```
define(N, 100)
...
if (i > N)
```

defines `N` to be `100` and uses the "symbolic constant" `N` in a later `if` statement. As noted, the left parenthesis must immediately follow the word `define` to signal that `define()` has arguments. If the macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. In the previous example, then, `N` is a macro with no arguments.

A macro name is only recognized as such if it appears surrounded by characters which cannot be used in a macro name. In the following example

```
define(N, 100)
...
if (NNN > 100)
```

the variable `NNN` is unrelated to the defined macro `N` even though the variable contains `N`s.

`m4` expands macro names into their defining text as soon as possible.

```
define(N, 100)
define(M, N)
```

defines `M` to be `100` because the string `N` is immediately replaced by `100` as the arguments of `define(M, N)` are collected. To put this another way, if `N` is redefined, `M` keeps the value `100`.

There are two ways to avoid this behavior. The first, which is specific to the situation described here, 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 the value of `M` is requested later, the result will always be the value of `N` at that time (because the M will be replaced by `N` which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of `define()` by quoting them. Any text surrounded by left and right single quotes is not expanded immediately, but has the quotes stripped off as the arguments are collected. The value of the quoted string is the string stripped of the quotes.

```
define(N, 100)
define(M, ‘N’)
```

defines `M` as the string `N`, not `100`.

The general rule is that **`m4`** always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word `define` is to appear in the output, the word must be quoted in the input:

```
‘define’ = 1;
```

It’s usually best to quote the arguments of a macro to assure that what you are assigning to the macro name actually gets assigned. To redefine `N`, for example, you delay its evaluation by quoting:

```
define(N, 100)
...
define(‘N’, 200)
```

Otherwise

```
define(N, 100)
...
define(N, 200)
```

the `N` in the second definition is immediately replaced by `100`. The effect is the same as saying

```
define(100, 200)
```

Note that this statement will be ignored by **`m4`** because only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed with the built-in macro `changequote()`:

```
changequote([, ])
```

In this example the macro makes the “quote” characters the left and right brackets instead of the left and right single quotes. The quote symbols can be up to five characters long. The original characters can be restored by using `changequote()` without arguments:

```
changequote
```

`undefine()` removes the definition of a macro or built-in:

```
undefine(‘N’)
```

Here the macro removes the definition of `N`. Be sure to quote the argument to `undefine()`. Built-ins can be removed with `undefine()` as well:

```
undefine(‘define’)
```

Note that once a built-in is removed or redefined, its original definition cannot be reused.

Macros can be renamed with `defn()`. Suppose you want the built-in `define()` to be called `XYZ()`. You specify

```
define(XYZ, defn(‘define’))
undefine(‘define’)
```

After this, `XYZ()` takes on the original meaning of `define()`.

```
XYZ(A, 100)
```

defines `A` to be `100`.

The built-in `ifdef()` provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef(‘pdp11’, ‘define(wordsize,16)’)
ifdef(‘u3b’, ‘define(wordsize,32)’)
```

The `ifdef()` macro permits three arguments. If the first argument is defined, the value of `ifdef()` is the second argument. If the first argument is not defined, the value of `ifdef()` is the third argument:

```
ifdef(‘unix’, on UNIX, not on UNIX)
```

If there is no third argument, the value of `ifdef()` is null.

Arguments

The previous sections focused on the simplest form of macro processing — replacing one string with another (fixed) string. Macros can also be defined so that different invocations have different results. In 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. The macro `bump()`, defined as

```
define(bump, $1 = $1 + 1)
```

is equivalent to `x = x + 1` for `bump(x)`.

A macro can have as many arguments as you want, but only the first nine are accessible individually, `$1` through `$9`. `$0` refers to the macro name itself. Arguments that are not supplied are replaced by null strings, so a macro can be defined that simply concatenates its arguments:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

`cat(x, y, z)` is equivalent to `xyz`. Arguments `$4` through `$9` are null because no corresponding arguments were provided.

Leading unquoted blanks, tabs, or new-lines that occur during argument collection are discarded. All other white space is retained, so

```
define(a,   b   c)
```

defines `a` to be `b   c`.

Arguments are separated by commas. A comma “protected” by parentheses does not terminate an argument:

```
define(a, (b,c))
```

has two arguments, `a` and `(b,c)`. You can specify a comma or parenthesis as an argument by quoting it.

`$*` is replaced by a list of the arguments given to the macro in a subsequent invocation. The listed arguments are separated by commas.

```
define(a, 1)
define(b, 2)
define(star, ‘$*’)
star(a, b)
```

gives the result `1,2`.

```
star(‘a’, ‘b’)
```

gives the same result because **`m4`** strips the quotes from `a` and `b` as it collects the arguments of `star()`, then expands `a` and `b` when it evaluates `star()`.

`$@` is identical to `$*` except that each argument in the subsequent invocation is quoted.

```
define(a, 1)
define(b, 2)
define(at, ‘$@’)
at(‘a’, ‘b’)
```

gives the result `a,b` because the quotes are put back on the arguments when `at()` is evaluated.

`$#` is replaced by the number of arguments in the subsequent invocation.

```
define(sharp, ‘$#’)
sharp(1, 2, 3)
```

gives the result `3`,

```
sharp()
```

gives the result `1`, and

```
sharp
```

gives the result `0`.

The built-in `shift()` returns all but its first argument. The other arguments are quoted and pushed back onto the input with commas in between. The simplest case

```
shift(1, 2, 3)
```

gives `2,3`. As with `$@`, you can delay the expansion of the arguments by quoting them, so

```
define(a, 100)
define(b, 200)
shift(‘a’, ‘b’)
```

gives the result `b` because the quotes are put back on the arguments when `shift()` is evaluated.

Arithmetic Built-Ins

`m4` provides three built-in macros for doing integer arithmetic. `incr()` increments its numeric argument by 1. `decr()` decrements by 1. To handle the common programming situation in which a variable is to be defined as “one more than N” you would use

```
define(N, 100)
define(N1, ‘incr(N)’)
```

`N1` is defined 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. Its operators in decreasing order of precedence are

```
+ -  (unary)
**
* / %
+ -
== != < <= > >=
! ~
&
| ^
&&
||
```

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 on the UNIX operating system.

As a simple example, you can define `M` to be `2**N+1` with

```
define(M, ‘eval(2**N+1)’)
```

Then the sequence

```
define(N, 3)
M(2)
```

gives `9` as the result.

File Inclusion

A new file can be included in the input at any time with the built-in macro `include()`:

```
include(filename)
```

inserts the contents of *filename* in place of the macro and its argument. The value of `include()` (its replacement text) is the contents of the file. If needed, the contents can be captured in definitions and so on.

A fatal error occurs if the file named in `include()` cannot be accessed. To get some control over this situation, the alternate form `sinclude()` ("silent include") can be used. This built-in says nothing and continues if the file named cannot be accessed.

Diversions

`m4` output can be diverted to temporary files during processing, and the collected material can be output on command. **`m4`** maintains nine of these diversions, numbered 1 through 9. If the built-in macro `divert(`*n*`)` is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the `divert()` or `divert(0)` macros, which resume the normal output process.

Diverted text is normally output at the end of processing in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in `undivert()` brings back all diversions in numerical order; `undivert()` with arguments brings back the selected diversions in the order given. "Undiverting" discards the diverted text (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. The current output stream is 0 during normal processing.

System Command

Any program can be run by using the `syscmd()` built-in:

```
syscmd(date)
```

invokes the UNIX operating system **`date`** command. Normally, `syscmd()` would be used to create a file for a subsequent `include()`.

To make it easy to name files uniquely, the built-in `maketemp()` replaces a string of `XXXXX` in the argument with the process ID of the current process.

Conditionals

Arbitrary conditional testing is performed with the built-in `ifelse()`. In its simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, `ifelse()` returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called `compare()` can be defined as one that compares two strings and returns `yes` or `no`, respectively, if they are the same or different:

```
define(compare, ‘ifelse($1, $2, yes, no)’)
```

Note the quotes, which prevent evaluation of `ifelse()` from occurring too early. If the final argument is omitted, the result is null, so

`ifelse(`*a, b, c*`)`

is *c* if *a* matches *b*, and null otherwise.

`ifelse()` can actually have any number of arguments and provides a limited form of multi-way decision capability. In the input

`ifelse(`*a, b, c, d, e, f, g*`)`

if the string `a` matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*.

String Manipulation

The `len()` macro returns the length of the string (number of characters) in its argument.

```
len(abcdef)
```

is `6`, and

```
len((a,b))
```

is `5`.

The `substr()` macro can be used to produce substrings of strings. If you type

substr(s, i, n)

it will return the substring of *s* that starts at the *i*th position (origin 0) and is *n* characters long. If *n* is omitted, the rest of the string is returned. If you type

```
substr(‘now is the time’,1)
```

it will return the following string:

```
now is the time
```

If *i* or *n* are out of range, a blank line is returned. For example, if you type

```
substr(‘now is the time’,-1)
```

or

```
substr(‘now is the time’,1,39)
```

you will get a blank line.

The `index(s1, s2)` macro returns the index (position) in `s1` where the string `s2` occurs, -1 if it does not occur. As with `substr()`, the origin for strings is 0.

`translit()` performs character transliteration and has the general form

    translit(s, f, t)

which modifies *s* by replacing any character in *f* by the corresponding character in *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*.

```
translit(s, aeiou)
```

would delete vowels from `s`.

The macro `dnl()` deletes all characters that follow it up to and including the next new-line. It is useful mainly for throwing away empty lines that otherwise would clutter up **`m4`** output. Using input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new-line at the end of each line that is not part of the definition. The new-line is copied into the output where it may not be wanted. When you add `dnl()` to each of these lines, the new-lines will disappear. Another method of achieving the same result is to input

```
divert(-1)
define(...)
...
divert
```

Printing

The built-in `errprint()` writes its arguments out on the standard error file. An example would be

```
errprint(‘fatal error’)
```

`dumpdef()` is a debugging aid that dumps the current names and definitions of items specified as arguments. If no arguments are given, then all current names and definitions are printed.

6
Lexical Analysis with lex

6
Lexical Analysis with lex

Introduction

lex is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you might check the spelling of words for errors; in code enciphering, you might translate certain patterns of characters into others; and in compiler writing, you might determine what the tokens are in the program to be compiled. The task common to all these problems is lexical analysis: recognizing different strings of characters that satisfy certain characteristics. Hence the name **lex**.

You don't have to use **lex** to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what **lex** does is produce such C programs. (**lex** is therefore called a program generator.) What **lex** offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using **lex** considerably outweigh it.

lex can also be used to collect statistical data on features of an input text, such as character count, word length, number of occurrences of a word, and so forth. In the remaining sections of this chapter, we will see

- How to generate a lexical analyzer program
- How to write **lex** source
- How to translate source
- How to use **lex** with **yacc**

Generating a Lexical Analyzer Program

lex generates a C language scanner from a source specification that you write to solve the problem at hand. This specification consists of a list of rules indicating sequences of characters — expressions — to be searched for in an input text, and the actions to take when an expression is found. We'll show you how to write a **lex** specification in “Writing lex Source” on page 6-3.

The C source code for the lexical analyzer is generated when you enter

```
lex lex.l
```

where **lex.l** is the file containing your **lex** specification. (The name **lex.l** is the favored convention, but you may use whatever name you want. Keep in mind, though, that the **.l** suffix is a convention recognized by other UNIX system tools, in particular, **make**.) The source code is written to an output file called **lex.yy.c** by default. That file contains the definition of a function called `yylex()` that returns 1 whenever an expression you have specified is found in the input text, 0 when end of file is encountered. Each call to `yylex()` parses one token. When `yylex()` is called again, it picks up where it left off.

Note that running **lex** on a specification that is spread across several files

```
lex lex1.l lex2.l lex3.l
```

produces one **lex.yy.c**. Invoking **lex** with the **-t** option causes it to write its output to `stdout` rather than **lex.yy.c**, so that it can be redirected:

```
lex -t lex.l > lex.c
```

Options to **lex** must appear between the command name and the file name argument.

The lexical analyzer code stored in **lex.yy.c** (or the **.c** file to which it was redirected) must be compiled to generate the executable object program, or scanner, that performs the lexical analysis of an input text. The **lex** library, **libl.a**, supplies a default `main()` that calls the function `yylex()`, so you need not supply your own `main()`. The library is accessed by specifying **libl** with the **-l** option to **cc**:

```
cc lex.yy.c -ll
```

Alternatively, you may want to write your own driver. The following is similar to the library version:

```
extern int yylex();

int yywrap()
{
    return(1);
}

main()
{
    while (yylex())
        ;
}
```

We'll take a closer look at the function `yywrap()` in “lex Routines” on page 6-10. For now it's enough to note that when your driver file is compiled with **lex.yy.c**

```
cc lex.yy.c driver.c
```

its `main()` will call `yylex()` at run time exactly as if the **lex** library had been loaded. The resulting executable reads `stdin` and writes its output to `stdout`. Figure 6-1 shows how **lex** works.

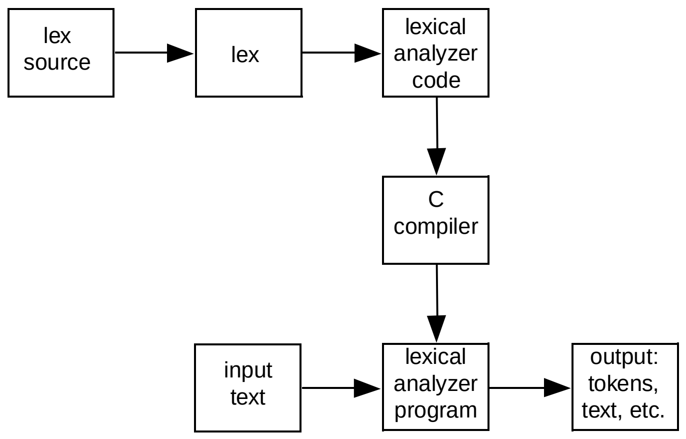


Figure 6-1. Creation and Use of a Lexical Analyzer with lex

Writing lex Source

`lex` source consists of at most three sections: definitions, rules, and user-defined routines. The rules section is mandatory. Sections for definitions and user routines are optional, but if present, must appear in the indicated order:

definitions
%%
rules
%%
user routines

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter %%. If a routines section follows, another %% delimiter ends the rules section. The %% delimiters must be entered at the beginning of a line, without leading blanks. If there is no second delimiter, the rules section is presumed to continue to the end of the program. Lines in the rules section that begin with white space and that appear before the first rule are copied to the beginning of the function `yylex()`, immediately after the first brace. You might use this feature to declare local variables for `yylex()`.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. The specification of the pattern must be entered at the beginning of a line. The scanner writes input that does not match a pattern directly to the output file. So the simplest lexical analyzer program is just the beginning rules delimiter, %%. It writes out the entire input to the output with no changes at all.

Regular Expressions

You specify the patterns you are interested in with a notation called a regular expression. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all:

```
apple
orange
pluto
```

These three regular expressions match any occurrences of those character strings in an input text. If you want to have the scanner remove every occurrence of `orange` from the input text, you could specify the rule

```
orange ;
```

Because you specified a null action on the right with the semicolon, the scanner does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string `orange` at all.

Operators

Unlike `orange` above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite. Thanks to the use of operators — summarized in Table 6-1 below — we can form regular expressions to signify any expression of a certain class. The `+` operator, for instance, means one or more occurrences of the preceding expression, the `?` means 0 or 1 occurrence(s) of the preceding expression (which is equivalent, of course, to saying that the preceding expression is optional), and `*` means 0 or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. We will see an example in a moment.) So `m+` is a regular expression that matches any string of `m`s:

```
mmm
m
mmmmm
```

and `7*` is a regular expression that matches any string of zero or more `7`s:

```
77
77777

777
```

The empty third line matches because it has no `7`s in it at all.

The `|` operator indicates alternation, so that `ab|cd` matches either `ab` or `cd`. The operators `{}` specify repetition, so that `a{1,5}` looks for 1 to 5 occurrences of `a`. Brackets, `[]`, indicate any one character from the string of characters specified between the brackets. Thus, `[dgka]` matches a single `d`, `g`, `k`, or `a`. Note that the characters between brackets must be adjacent, without spaces or punctuation. The `^` operator, when it appears as the first character after the left bracket, indicates all characters in the standard set except those specified between the brackets. (Note that `|`, `{}`, and `^` may serve other purposes as well; see below.) Ranges within a standard alphabetic or numeric order (`A` through `Z`, `a` through `z`, `0` through `9`) are specified with a hyphen. `[a-z]`, for instance, indicates any lowercase letter. Somewhat more interestingly,

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether upper or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize `*`, `&`, `r`, and `#`, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands. If you want to include the hyphen character in the class, it should appear as the first or last character in the brackets: `[-A-Z]` or `[A-Z-]`.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a `*`, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the `*`, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

```
e
not
idenTIFIER
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers:

```
not_idenTIFIER
5times
$hello
```

because `not_idenTIFIER` has an embedded underscore; `5times` starts with a digit, not a letter; and `$hello` starts with a special character.

A potential problem with operator characters is how we can specify them as characters to look for in a search pattern. The last example, for instance, will not recognize text with a `*` in it. **`lex`** solves the problem in one of two ways: an operator character preceded by a backslash, or characters (except backslash) enclosed in double quotation marks, are taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, a `*` followed by any number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a `\` itself, we need two backslashes: `\\`. Similarly, `“x\*x”` matches `x*x`, and `“y“ z“` matches `y”z`. Other **`lex`** operators are noted as they arise in the discussion below. **`lex`** recognizes all the C language escape sequences.

Table 6-1. lex Operators

Expression	Description
`\`*x*	*x*, if *x* is a **`lex`** operator
“*xy*”	*xy*, even if *x* or *y* are **`lex`** operators (except `\`)
`[`*xy*`]`	*x* or *y*
`[`*x*-*z*`]`	*x*, *y*, or *z*
`[^`*x*`]`	any character but *x*
`.`	any character but new-line
`^`*x*	*x* at the beginning of a line
`<`*y*`>`*x*	*x* when **`lex`** is in start condition *y*
x`$`	*x* at the end of a line
x`?`	optional *x*
x`*`	0, 1, 2, . . . instances of *x*
x`+`	1, 2, 3, . . . instances of *x*
x`{`*m*`,`*n*`}`	*m* through *n* occurrences of *x*
xx`|`*yy*	either *xx* or *yy*
x `|`	the action on *x* is the action for the next rule
`(`*x*`)`	*x*
x`/`*y*	*x* but only if followed by *y*
`{`*xx*`}`	the translation of *xx* from the definitions section

Actions

Once the scanner recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. You supply the actions. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. You write these actions as program fragments in C. An action may consist of as many statements as are needed for the job at hand. You may want to change the text in some way or simply print a message noting that the text has been found. So, to recognize the expression Amelia Earhart and to note such recognition, the rule

```
"Amelia Earhart"   printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize the ends of lines and increment a line counter. As we have noted, **lex** uses the standard C escape sequences, including `\n` for new-line. So, to count lines we might have

```
\n    lineno++;
```

where `lineno`, like other C variables, is declared in the definitions section that we discuss later.

Input is ignored when the C language null statement `;` is specified. So the rule

```
[ \t\n] ;
```

causes blanks, tabs, and new-lines to be ignored. Note that the alternation operator `|` can also be used to indicate that the action for a rule is the action for the next rule. The previous example could have been written:

```
" "    |
\t     |
\n     ;
```

with the same result.

The scanner stores text that matches an expression in a character array called `yytext[]`. You can print or manipulate the contents of this array as you like. In fact, **lex** provides a macro called `ECHO` that is equivalent to `printf("%s", yytext)`. We'll see an example of its use in "Start Conditions" on page 6-13.

Sometimes your action may consist of a long C statement, or two or more C statements, and you wish to write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings, and print out each one as soon as it is found, your **lex** code might be

```
\+?[1-9]+           { digstrngcount++;
                     printf("%d",digstrngcount);
                     printf("%s", yytext);   }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the `?` indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign will match the specification. "Advanced lex Usage" explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that let you process input text riddled with quite complicated patterns. These include rules that decide what specification is relevant when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining an example that draws together several of the points already covered:

```
%%
-[0-9]+          printf("negative integer");
\+?[0-9]+        printf("positive integer");
-0.[0-9]+        printf("negative fraction, no whole number part");
rail[ \t]+road   printf("railroad is one word");
crook            printf("Here's a crook");
function         subprogcount++;
G[a-zA-Z]*       { printf("may have a G word here:%s", yytext);
                 Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and -1. The use of the terminating `+` in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for `railroad` matches cases where one or more blanks intervene between the two syllables of the word. In the cases of `railroad` and `crook`, we could have simply printed a synonym rather than the messages stated. The rule recognizing a function simply increments a counter. The last rule illustrates several points:

- The braces specify an action sequence that extends over several lines.
- Its action uses the **`lex`** array `yytext[]`, which stores the recognized character string.
- Its specification uses the `*` to indicate that zero or more letters may follow the `G`.

Some Special Features

Besides storing the matched input text in `yytext[]`, the scanner automatically counts the number of characters in a match and stores it in the variable `yyleng`. You may use this variable to refer to any specific character just placed in the array `yytext[]`. Remember that C language array indexes start with 0, so to print out the third digit (if there is one) in a just recognized integer, you might enter

```
[1-9]+         {if (yyleng > 2)
               printf("%c", yytext[2]); }
```

`lex` follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. In the following lexical analyzer example, the “reserved word” `end` could match the second rule as well as the eighth, the one for identifiers:

```
begin                           return(BEGIN);
end                             return(END);
while                           return(WHILE);
if                              return(IF);
package                         return(PACKAGE);
reverse                         return(REVERSE);
loop                            return(LOOP);
[a-zA-Z][a-zA-Z0-9]*            { tokval = put_in_tabl();
                                return(IDENTIFIER); }
[0-9]+                          { tokval = put_in_tabl();
                                return(INTEGER); }
\+                              { tokval = PLUS;
                                return(ARITHOP); }
\-                              { tokval = MINUS;
                                return(ARITHOP); }
>                               { tokval = GREATER;
                                return(RELOP); }
>=                              { tokval = GREATEREQL;
                                return(RELOP); }
```

`lex` follows the rule that, where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed. By placing the rule for `end` and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize `>` and `>=`. If the text has the string `>=` at one point, you might worry that the lexical analyzer would stop as soon as it recognized the `>` character and execute the rule for `>`, rather than read the next character and execute the rule for `b`. **`lex`** follows the rule that it matches the longest character string possible and executes the rule for that. Here the scanner would recognize the `>=` and act accordingly. As a further example, the rule would enable you to distinguish `+` from `++` in a C program.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure that you've in fact found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the `DO` statement in Fortran. In the statement

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index `k` until we read the first comma. Until then, we might have the assignment statement

```
DO50k = 1
```

(Remember that Fortran ignores all blanks.) The way to handle this is to use the slash, `/`, which signifies that what follows is trailing context, something not to be stored in `yytext[]`, because it is not part of the pattern itself. So the rule to recognize the Fortran `DO` statement could be

```
DO/([ ]*[0-9]+[ ]*[a-zA-Z0-9]+=[a-zA-Z0-9]+,) {
    printf("found DO");
    }
```

Different versions of Fortran have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length. See "Start Conditions" on page 6-13 for a discussion of **lex**`s similar handling of prior context.

lex uses the `$` symbol as an operator to mark a special trailing context — the end of a line. An example would be a rule to ignore all blanks and tabs at the end of a line:

```
[  \t]+$     ;
```

which could also be written:

```
[  \t]+/\n   ;
```

On the other hand, if you want to match a pattern only when it starts a line or a file, you can use the `^` operator. Suppose a text-formatting program requires that you not start a line with a blank. You might want to check input to the program with some such rule as

```
^[ ]      printf("error: remove leading blank");
```

Note the difference in meaning when the `^` operator appears inside the left bracket, as described in "Operators" on page 6-4.

lex Routines

Some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three macros to handle these tasks — `input()`, `unput(c)`, and `output(c)`, respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use `input()`, thus:

```
\"          while (input() != '"');
```

Upon finding the first double quotation mark, the scanner will simply continue reading all subsequent characters so long as none is a double quotation mark, and not look for a match again until it finds a second double quotation mark. (See the further examples of `input()` and `unput(c)` usage in "User Routines" on page 6-14.)

By default, these routines are provided as macro definitions. To handle special I/O needs, such as writing to several files, you may use standard I/O routines in C to rewrite the functions. Note, however, that they must be modified consistently. In particular, the character set used must be consistent in all routines, and a value of 0 returned by `input()` must mean end of file. The relationship between `input()` and `unput(c)` must be maintained or the **lex** lookahead will not work.

If you do provide your own `input()`, `output(c)`, or `unput(c)`, you will have to write a `#undef input` and so on in your definitions section first:

```
#undef input
#undef output
    . . .
#define input()  . . . etc.
more declarations
    . . .
```

Your new routines will replace the standard ones. See “Definitions” on page 6-12 for further details.

A **lex** library routine that you may sometimes want to redefine is `yywrap()`, which is called whenever the scanner reaches end of file. If `yywrap()` returns 1, the scanner continues with normal wrapup on end of input. Occasionally, however, you may want to arrange for more input to arrive from a new source. In that case, redefine `yywrap()` to return 0 whenever further processing is required. The default `yywrap()` always returns 1. Note that it is not possible to write a normal rule that recognizes end of file; the only access to that condition is through `yywrap()`. Unless a private version of `input()` is supplied, a file containing nulls cannot be handled because a value of 0 returned by `input()` is taken to be end of file.

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include `yymore()`, `yyless(n)`, and `REJECT`. Recall that the text that matches a given specification is stored in the array `yytext[]`. In general, once the action is performed for the specification, the characters in `yytext[]` are overwritten with succeeding characters in the input stream to form the next match. The function `yymore()`, by contrast, ensures that the succeeding characters recognized are appended to those already in `yytext[]`. This lets you do one thing and then another, when one string of characters is significant and a longer one including the first is significant as well. Consider a language that defines a string as a set of characters between double quotation marks and specifies that to include a double quotation mark in a string it must be preceded by a backslash. The regular expression matching that is somewhat confusing, so it might be preferable to write:

```
\" [^"]*{
        if (yytext[yyleng-1] == '\\')
            yymore();
        else
        . . . normal processing
        }
```

When faced with the string `“abc“ def”`, the scanner will first match the characters `“abc,` whereupon the call to `yymore()` will cause the next part of the string `“def` to be tacked on the end. The double quotation mark terminating the string should be picked up in the code labeled “normal processing.”

The function `yyless(n)` lets you specify the number of matched characters on which an action is to be performed: only the first `n` characters of the expression are retained in `yytext[]`. Subsequent processing resumes at the `n`th + 1 character. Suppose you are again in the code deciphering business and the idea is to work with only half the characters in a sequence that ends with a certain one, say upper or lowercase `Z`. The code you want might be

```
[a-yA-Y]+[Zz]    {  yyless(yyleng/2);
                    . . . process first half of string . . . }
```

Finally, the function `REJECT` lets you more easily process strings of characters even when they overlap or contain one another as parts. `REJECT` does this by immediately jumping to the next rule and its specification without changing the contents of `yytext[]`. If you want to count the number of occurrences both of the regular expression `snapdragon` and of its subexpression `dragon` in an input text, the following will do:

```
    snapdragon     {countflowers++; REJECT;}
    dragon         countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions `comedian` and `diana`, even where the input text has sequences such as `comediana.`:

```
    comedian      {comiccount++; REJECT;}
    diana         princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, you declare the counters and other necessary variables in the definitions section commencing the **`lex`** specification.

Definitions

The **`lex`** definitions section may contain any of several classes of items. The most critical are external definitions, preprocessor statements like **`#include`**, and abbreviations. Recall that for valid **`lex`** source this section is optional, but in most cases some of these items are necessary. Preprocessor statements and C source code should appear between a line of the form `%{` and one of the form `%}`. All lines between these delimiters — including those that begin with white space — are copied to **`lex.yy.c`** immediately before the definition of `yylex()`. (Lines in the definition section that are not enclosed by the delimiters are copied to the same place provided they begin with white space.) The definitions section is where you would normally place C definitions of objects accessed by actions in the rules section or by routines with external linkage.

One example occurs in using **`lex`** with **`yacc`**, which generates parsers that call a lexical analyzer. In this context, you should include the file **`y.tab.h`**, which may contain **`#define`**s for token names:

```
    %{
    #include "y.tab.h"
    extern int tokval;
    int lineno;
    %}
```

After the `%}` that ends your **`#include`**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within braces. Abbreviations avoid needless repetition in writing your specifications and make them easier to read.

As an example, reconsider the **`lex`** source reviewed at the beginning of this section on advanced **`lex`** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:

```
D               [0-9]
L               [a-zA-Z]
B               [ \t]+
%%
-{D}+           printf("negative integer");
\+?{D}+         printf("positive integer");
-0.{D}+         printf("negative fraction");
G{L}*           printf("may have a G word here");
rail{B}road     printf("railroad is one word");
crook           printf("criminal");
  .              .
  .              .
```

Start Conditions

Some problems require for their solution a greater sensitivity to prior context than is afforded by the `^` operator alone. You may want different rules to be applied to an expression depending on a prior context that is more complex than the end of a line or the start of a file. In this situation you could set a flag to mark the change in context that is the condition for the application of a rule, then write code to test the flag. Alternatively, you could define for **`lex`** the different “start conditions” under which it is to apply each rule.

Consider this problem: copy the input to the output, except change the word `magic` to the word `first` on every line that begins with the letter `a`; change `magic` to `second` on every line that begins with `b`; change `magic` to `third` on every line that begins with `c`. Here is how the problem might be handled with a flag. Recall that `ECHO` is a **`lex`** macro equivalent to `printf(“%s”, yytext)`:

```
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;
    }
}
```

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 are referenced at the head of a rule with `<>` brackets. So

<name1>expression

is a rule that is only recognized when the scanner is in 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;
```

resets the initial condition of the scanner. A rule may be active in several start conditions. That is,

`<`*name1*`,`*name2*`,`*name3*`>`

is a valid prefix. Any rule not beginning with the `<>` prefix operators is always active.

The example can be written with start conditions as follows:

```
%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");
```

User Routines

You may want to use your own routines in **`lex`** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function `put_in_tabl()`, to be discussed in “Using lex with yacc” on page 6-15, is a good candidate for the user routines section of a **`lex`** specification.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between `/*` and `*/`:

```
%{
static skipcmnts();
%}
%%
"/*"                  skipcmnts();
.
.               /* rest of rules */
%%
static
skipcmnts()
{
       for(;;)
       {
          while (input() != '*')
         ;
          if (input() != '/')
                unput(yytext[yyleng-1])
          else return;
       }
 }
```

There are three points of interest in this example. First, the `unput(c)` macro (putting back the last character read) is necessary to avoid missing the final `/` if the comment ends with a `**/`. In this case, eventually having read a `*`, the scanner finds that the next character is not the terminal `/` and must read some more. Second, the expression `yytext[yyleng-1]` picks out that last character read. Third, this routine assumes that the comments are not nested, which is indeed the case with the C language.

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system tool **`yacc`** (see Chapter 7 (“Parsing with yacc”)). **`yacc`** generates parsers, programs that analyze input to insure that it is syntactically correct. **`lex`** often forms a fruitful union with **`yacc`** in the compiler development context. Whether or not you plan to use **`lex`** with **`yacc`**, be sure to read this section because it covers information of interest to all **`lex`** programmers.

As noted, a program uses the **`lex`**-generated scanner by repeatedly calling the function `yylex()`. This name is used because a **`yacc`**-generated parser calls its lexical analyzer with this very name. To use **`lex`** to create the lexical analyzer for a compiler, you want to end each **`lex`** action with the statement `return` *token*, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, called `yyparse()` by **`yacc`**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operator, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant is, say, `9` or `888`, whether the operator is `+` or `*`, and whether the relational operator is `=` or `>`. Consider the following portion of **lex** source (discussed in another context earlier) for a scanner that recognizes tokens in a “C-like” language:

```
begin                          return(BEGIN);
end                            return(END);
while                          return(WHILE);
if                             return(IF);
package                        return(PACKAGE);
reverse                        return(REVERSE);
loop                           return(LOOP);
[a-zA-Z][a-zA-Z0-9]*           { tokval = put_in_tabl();
                               return(IDENTIFIER); }
[0-9]+                         { tokval = put_in_tabl();
                               return(INTEGER); }
\+                             { tokval = PLUS;
                               return(ARITHOP); }
\-                             { tokval = MINUS;
                               return(ARITHOP); }
>                              { tokval = GREATER;
                               return(RELOP); }
>=                             { tokval = GREATEREQL;
                               return(RELOP); }
```

Despite appearances, the tokens returned, and the values assigned to `tokval`, are indeed integers. Good programming style dictates that we use informative terms such as `BEGIN`, `END`, `WHILE`, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using `#define` statements in your parser calling routine in C. For example,

```
#define BEGIN 1
#define END 2
.
#define PLUS 7
.
```

If the need arises to change the integer for some token type, you then change the `#define` statement in the parser rather than hunt through the entire program changing every occurrence of the particular integer. In using **yacc** to generate your parser, insert the statement

```
#include "y.tab.h"
```

in the definitions section of your **lex** source. The file **y.tab.h**, which is created when **yacc** is invoked with the **-d** option, provides `#define` statements that associate token names such as `BEGIN`, `END`, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable `tokval`. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable `yylval` for the same purpose.

Note that the example shows two ways to assign a value to `tokval`. First, a function `put_in_tabl()` places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, `put_in_tabl()` assigns a type value to `tokval` so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function `put_in_tabl()` would be a routine that the compiler writer might place in the user routines section of the parser. Second, in the last few actions of the example, `tokval` is assigned a specific integer indicating which arithmetic or relational operator the scanner recognized. If the variable `PLUS`, for instance, is associated with the integer 7 by means of the `#define` statement above, then when a `+` is recognized, the action assigns to `tokval` the value 7, which indicates the `+`. The scanner indicates the general class of operator by the value it returns to the parser (that is, the integer signified by `ARITHOP` or `RELOP`).

In using **`lex`** with **`yacc`**, either may be run first. The command

```
yacc -d grammar.y
```

generates a parser in the file **`y.tab.c`**. As noted, the **`-d`** option creates the file **`y.tab.h`**, which contains the `#define` statements that associate the **`yacc`**-assigned integer token values with the user-defined token names. Now you can invoke **`lex`** with the command

```
lex lex.l
```

then compile and link the output files with the command

```
cc lex.yy.c y.tab.c -ly -ll
```

Note that the **`yacc`** library is loaded (via **`-ly`**) before the **`lex`** library (via **`-ll`**) to ensure that the supplied `main()` will call the **`yacc`** parser.

Miscellaneous

Recognition of expressions in an input text is performed by a deterministic finite automaton generated by **`lex`**. The **`-v`** option prints out for you a small set of statistics describing the finite automaton. (For a detailed account of finite automata and their importance for **`lex`**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

`lex` uses a table to represent its finite automaton. The maximum number of states that the finite automaton allows is set by default to 500. If your **`lex`** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing another entry in the definitions section of your **`lex`** source as follows:

```
%n 700
```

This entry tells **`lex`** to make the table large enough to handle as many as 700 states. (The **`-v`** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is `a`, thus:

```
%a 2800
```

Summary of Source Format

- The general form of a **`lex`** source file is

  definitions
  `%%`
  rules
  `%%`
  user routines

- The definitions section contains any combination of

  - definitions of abbreviations in the form

    name space translation

  - included code in the form

    ```
    %{
    C code
    %}
    ```

  - start conditions in the form

    `Start` *name1 name2* `. . .`

  - 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:

`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.

- The **`lex`** operator characters are

  ```
  " \ [] ^ - ? . * | () $ / {} <> +
  ```

- Important **`lex`** variables, functions, and macros are

`yytext[]`	array of `char`
`yyleng`	`int`
`yylex()`	function
`yywrap()`	function
`yymore()`	function
`yyless(`*n*`)`	function
`REJECT`	macro
`ECHO`	macro
`input()`	macro
`unput(`*c*`)`	macro
`output(`*c*`)`	macro

7
Parsing with yacc

7 Parsing with yacc

Introduction

yacc provides a general tool for imposing structure on the input to a computer program. When you use **yacc**, you prepare a specification that includes

- A set of rules to describe the elements of the input;
- Code to be invoked when a rule is recognized;
- Either a definition or declaration of a low-level scanner to examine the input.

yacc then turns the specification into a C language function that examines the input stream. This function, called a *parser*, works by calling the low-level scanner. The scanner, called a *lexical analyzer*, picks up items from the input stream. The selected items are known as *tokens*. Tokens are compared to the input construct rules, called *grammar rules*. When one of the rules is recognized, the code you have supplied for the rule is invoked. This code is called an *action*. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be

```
date : month_name day ´,´ year   ;
```

where `date`, `month_name`, `day`, and `year` represent constructs of interest; presumably, `month_name`, `day`, and `year` are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means 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 evaluating the input. With proper definitions, the input

```
July  4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as non-terminal symbols. To avoid confusion, we will refer to terminal symbols as *tokens*.

There is considerable leeway in deciding whether to recognize constructs 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. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a `month_name` is seen. In this case, `month_name` is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan, input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. 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 often can 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 constructs that 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 program development.

The remainder of this chapter describes the following subjects:

- Basic process of preparing a **yacc** specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions

- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions to improve the style and efficiency of the specifications
- Advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or non-terminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs (%%; the percent sign is generally used in **yacc** specifications as an escape character).

A full specification file looks like

```
declarations
%%
rules
%%
subroutines
```

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest valid **yacc** specification might be

```
%%
S:;
```

Blanks, tabs, and new-lines are ignored, but they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is valid. They are enclosed in `/*` and `*/`, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where A represents a non-terminal symbol, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length and may be made up of letters, periods, underscores, and digits although a digit may not be the first character of a name. Upper case and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or non-terminal symbols.

A literal consists of a character enclosed in single quotes. As in the C language, the backslash is an escape character within literals. **yacc** recognizes all the C language escape sequences. For a number of technical reasons, the null character 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 is 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
    ;
```

by using the vertical bar. 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 more readable and easier to change.

If a non-terminal symbol matches the empty string, this can be indicated by

```
epsilon :   ;
```

The blank space following the colon is understood by **yacc** to be a non-terminal symbol named `epsilon`.

Names representing tokens must be declared. This is most simply done by writing

```
%token   name1  name2  name3
```

and so on in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminal symbols, the start symbol has particular importance. By default, the symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and 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 end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end of file or end of record.

Actions

With each grammar rule, you can associate actions to be performed when the rule is recognized. 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 language statement and as such can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in `{`and`}`. For example,

```
A   :  '('  B  ')'
    {
       hello( 1, "abc" );
    }
```

and

```
XXX   :  YYY  ZZZ
      {
         (void) printf("a message\n");
         flag = 25;
      }
```

are grammar rules with actions.

The `$` symbol is used to facilitate communication between the actions and the parser, The pseudo-variable `$$` represents the value returned by the complete action. For example, the action

```
{  $$ = 1;  }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action can use the pseudo-variables `$1, $2, . . . $n`. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is

```
A   :  B  C  D   ;
```

then `$2` has the value returned by `C`, and `$3` the value returned by `D`. The rule

```
expr   :   '('  expr  ')'   ;
```

provides a common example. One would expect the value returned by this rule to be the value of the `expr` within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by

```
expr   :    '('  expr  ')'
       {
            $$ = $2 ;
       }
```

By default, the value of a rule is the value of the first element in it (`$1`). Thus, grammar rules of the form

```
    A   :   B    ;
```

frequently need not have an explicit action. In previous examples, all the actions came at the end of 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 action 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 below the effect is to set `x` to 1 and `y` to the value returned by `C`:

```
    A   :   B
                {
                   $$ = 1;
                }
                C
        {
                x = $2;
                y = $3;
        }
        ;
```

Actions that do not terminate a rule are handled by **yacc** by manufacturing a new non-terminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered by recognizing this added rule. **yacc** treats the above example as if it had been written

```
    $ACT    :   /* empty */
            {
                $$ = 1;
            }
            ;
    A       :   B  $ACT  C
            {
                x = $2;
                y = $3;
            }
            ;
```

where `$ACT` is an empty action.

In many applications, output is not done directly by the actions. 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. Then a parse tree can be built by supplying actions such as

```
    expr    :   expr  '+'  expr
            {
                $$ = node( '+', $1, $3 );
            }
```

in the specification.

You may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in `%{` and `%}`. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
%{   int variable = 0;   %}
```

could be placed in the declarations section making `variable` accessible to all of the actions. You should avoid names beginning with `yy` because the **`yacc`** parser uses only such names. Note, too, that in the examples shown thus far all the values are integers. A discussion of values of other types is found in "Advanced Topics" on page 7-26. Finally, note that in the following case

```
%{
    int i;
    printf("%}");
%}
```

`yacc` will start copying after `%{` and stop copying when it encounters the first `%}`, the one in `printf()`. In contrast, it would copy `%{` in `printf()` if it encountered it there.

Lexical Analysis

You 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 the user. In either case, the `#define` mechanism of C language 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 the screen shown below to return the appropriate token.

```
int 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 a token number of `DIGIT` and a value equal to the numerical value of the digit. You put the lexical analyzer code in the subroutines section and the declaration for `DIGIT` in the declarations section. Alternatively, you can put the lexical analyzer code in a separately compiled file, provided

- You invoke **`yacc`** with the **`-d`** option, which generates a file called **`y.tab.h`** that contains `#define` statements for the tokens, and
- You `#include` **`y.tab.h`** in the separately compiled lexical analyzer.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names `if` or `while` 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.

In the default situation, token 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.

If you prefer to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **`yacc`**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker must have token number 0 or be negative. You cannot redefine this token number. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

As noted in Chapter 6 (“Lexical Analysis with lex”), lexical analyzers produced by **`lex`** are designed to work in close harmony with **`yacc`** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **`lex`** can be used to produce quite complicated lexical analyzers, but there remain some languages that do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

Parser Operation

`yacc` turns the specification file into a C language procedure, 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. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

The parser produced by **`yacc`** consists of a finite state machine with a stack. The parser is also capable of reading and remembering 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: `shift`, `reduce`, `accept`, and `error`. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. 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 of the stack and in the lookahead token being processed or left alone.

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.

The `reduce` action keeps the stack from growing without bounds. `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 or not to reduce. In fact, the default action (represented by `.`) is often a `reduce` action.

`reduce` actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads 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

```
    A  :  x  y  z   ;
```

is being reduced. 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 but is not affected by a `goto`. In any case, the uncovered state contains an entry such as

```
    A   goto 20
```

causing state 20 to be pushed onto the stack and become 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 stacks. The uncovered state is in fact the current state.

The `reduce` action is also important in the treatment of user-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 user 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`, and so on refer to the value stack.

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 end-marker 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 valid 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 discussed later.

Consider

```
    %token  DING  DONG  DELL
    %%
    rhyme   :   sound  place
            ;
    sound   :   DING  DONG
            ;
    place   :   DELL
            ;
```

as a **yacc** specification. When **yacc** is invoked with the **-v** (verbose) 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) follows.

```
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
```

The actions for each state are specified and 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. The following input

```
DING  DONG  DELL
```

can be used to track the operations of the parser. 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 and becomes the lookahead token. The action in state 0 on `DING` is `shift 3`, 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 and becomes the lookahead token. The action in state 3 on the token `DONG` is `shift 6`, 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

```
sound   :   DING  DONG
```

which is rule 2. 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. State 2 is pushed onto the stack and becomes 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 and the end-marker is obtained indicated by `$end` in the **`y.output`** file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

You might want 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 spent with this and other simple examples is repaid when problems arise in more complicated contexts.

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 completely 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. Given the input

```
expr  -  expr  -  expr
```

consider the problem that confronts the parser. When the parser has read the second `expr`, the input 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, if the parser sees

```
expr  -  expr
```

it could defer the immediate application of the rule and continue reading the input until

```
expr  -  expr  -  expr
```

is seen. It could then apply the rule to the rightmost three symbols, reducing them to `expr`, which results in

```
expr  -  expr
```

being left. 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 one of two valid things, shift or reduce. It 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 valid 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 the choice to make in a given situation is called a disambiguating rule.

`yacc` invokes two default disambiguating rules:

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 **`yacc`** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but `reduce-reduce` conflicts should be avoided when 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 rules 1 and 2 above.

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

```
stat   :   IF  '('  cond  ')'  stat
       |   IF  '('  cond  ')'  stat  ELSE  stat
       ;
```

which is a fragment from a programming language involving an `if-then-else` statement. In these rules, `IF` and `ELSE` are tokens, `cond` is a non-terminal symbol describing conditional (logical) expressions, and `stat` is a non-terminal 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 because input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
}
ELSE
      S2
```

or

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
      ELSE
            S2
}
```

where 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  (  C1  )  IF  (  C2  )  S1
```

and is looking at the `ELSE`. It can immediately reduce by the simple `if` rule to get

```
IF  (  C1  )  stat
```

and then read the remaining input

```
    ELSE  S2
```

and reduce

```
    IF  (  C1  )  stat  ELSE  S2
```

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, `S2` read, and then the right-hand portion of

```
    IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the `if-else` rule to get

```
    IF  (  C1  )  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 the one 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, such as

```
    IF  (  C1  )  IF  (  C2  )  S1
```

have already been seen. 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 **`-v`** output. 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
```

where the first line describes the conflict — giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underscore marks the portion of the grammar rules that 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
```

because the `ELSE` will have been shifted in this state. In state 23, the alternative action (specified by `.`) is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not `ELSE`, the parser reduces to

```
    stat  :  IF  '('  cond  ')'  stat
```

by grammar rule 18.

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, rule numbers are printed in parentheses after those rules that can be reduced. In most states, there is a `reduce` action possible, and `reduce` is the default command. If you encounter unexpected `shift-reduce` conflicts, you will probably want to look at the **`-v`** output to decide whether the default actions are appropriate.

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  exprand
```

and

```
    expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. You specify as disambiguating rules 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 the **`yacc`** keywords `%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. `+` and `-` are left associative and have lower precedence than `*` and `/`, which are also left associative. The keyword `%right` is used to describe right associative operators. The keyword

`%nonassoc` is used to describe operators, like the operator `.LT.` in Fortran, that may not associate with themselves. That is, because

```
A .LT. B .LT. C
```

is invalid in Fortran, `.LT.` 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) ) )
```

in order to achieve the correct precedence of operators. 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 minus.

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 causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr   :   expr  '+'  expr
       |   expr  '-'  expr
       |   expr  '*'  expr
       |   expr  '/'  expr
       |   '-'  expr      %prec  '*'
       |   NAME
       ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by `%left`, `%right`, and `%nonassoc` need not, but may, be declared by `%token` as well.

Precedences and associativities are used by **`yacc`** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. 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` or `shift-reduce` conflict, and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given in the preceding 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 precedences are equal, then associativity is used. Left associative implies `reduce`; right associative implies `shift`; “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 a cookbook fashion until some experience has been gained. The **`y.output`** file is useful in deciding whether the parser is actually doing what was intended.

To illustrate further how you might use the precedence keywords to resolve a `shift-reduce` conflict, we'll look at an example similar to the one described in the previous section. Consider the following C statement:

```
if (flag) if (anotherflag) x = 1;
else x = 2;
```

The problem for the parser is whether the `else` goes with the first or the second `if`. C programmers will recognize that the `else` goes with the second `if`, contrary to what the misleading indentation suggests. The following **`yacc`** grammar for an `if-then-else` construct abstracts the problem. That is, the input `iises` will model the C statement shown above.

```
%{
#include <stdio.h>
%}
%token SIMPLE IF ELSE
%%
S        : stmnt '\n'
         ;
stmnt    : SIMPLE
         | if_stmnt
         ;
if_stmnt : IF stmnt
             { printf("simple if\n");}
         | IF stmnt ELSE stmnt
             { printf("if_then_else\n");}
         ;
%%
int
yylex() {
    int c;
    c=getchar();
    if (c==EOF) return 0;
    else switch(c) {
         case 'i': return IF;
         case 's': return SIMPLE;
         case 'e': return ELSE;
         default: return c;
         }
}
```

When the specification is passed to **`yacc`**, however, we get the following message:

```
conflicts: 1 shift/reduce
```

The problem is that when **`yacc`** has read `iis` in trying to match `iises`, it has two choices: recognize `is` as a statement (reduce), or read some more input (shift) and eventually recognize `ises` as a statement.

One way to resolve the problem is to invent a new token `REDUCE` whose sole purpose is to give the correct precedence for the rules:

```
%{
#include <stdio.h>
%}
%token SIMPLE IF
%nonassoc REDUCE
%nonassoc ELSE
%%
S        : stmnt '\n'
         ;
stmnt    : SIMPLE
         | if_stmnt
         ;
if_stmnt : IF stmnt %prec REDUCE
             { printf("simple if");}
         | IF stmnt ELSE stmnt
             { printf("if_then_else");}
         ;
%%
...
```

Since the precedence associated with the second form of `if_stmnt` is higher now, **`yacc`** will try to match that rule first, and no conflict will be reported.

Actually, in this simple case, the new token is not needed:

```
%nonassoc IF
%nonassoc ELSE
```

would also work. Moreover, it is not really necessary to resolve the conflict in this way, because, as we have seen, **`yacc`** will shift by default in a `shift-reduce` conflict. Resolving conflicts is a good idea, though, in the sense that you should not see diagnostic messages for correct specifications.

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/or, 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 user some control over this process, **`yacc`** provides the token name `error`. This name can be used in grammar rules. In effect, it suggests where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token `error` is valid. 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
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might validly follow a statement, and starts 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, and so forth.

Error rules such as the above are very general but difficult to control. Rules such as

```
    stat   :   error  ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon 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. The following example

```
    input : error  '\n'
                 {
                    (void) printf("Reenter last line: " );
                 }
                 input
          {
            $$ = $4;
          }
          ;
```

is one way to do this. 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 force the parser to believe that error recovery has been accomplished. The statement

```
    yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input : error  '\n'
             {
                yyerrok;
                (void) printf("Reenter last line: " );
             }
             input
      {
         $$ = $4;
      }
      ;
```

As previously mentioned, 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 user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by `yylex()` is presumably the first token in a valid statement. The old invalid token must be discarded and the `error` state reset. A rule similar to

```
stat   :   error
       {
          resynch();
          yyerrok  ;
          yyclearin;
       }
       ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

You create a **`yacc`** parser with the command

```
yacc grammar.y
```

where **`grammar.y`** is the file containing your **`yacc`** specification. (The **`.y`** suffix is a convention recognized by other UNIX system commands. It is not strictly necessary.) The output is a file of C language subroutines called **`y.tab.c`**. The function produced by **`yacc`** is called `yyparse()`, and is integer-valued. When it is called, it in turn repeatedly calls `yylex()`, the lexical analyzer supplied by the user (see “Lexical Analysis” on page 7-7), to obtain input tokens. Eventually, an error is detected, `yyparse()` returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, `yyparse()` returns the value 0.

You must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a routine called `main()` must be defined that eventually calls `yyparse()`. In addition, a routine called `yyerror()` is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **`yacc`**, a library has been provided with default versions of `main()` and `yyerror()`. The library, `liby`, is accessed by a **`-ly`** argument to the **`cc`** command. The source codes

```
main()
{
     return (yyparse());
}
```

and

```
# include <stdio.h>
yyerror(s)
        char *s;
{
        (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs. The argument to `yyerror()` is a string containing an error message, usually the string `syntax error`. The average application wants 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. Since the `main()` routine is probably supplied by the user (to read arguments, for instance), the **`yacc`** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable `yydebug` is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using **`gdb(1).`**

Hints for Preparing Specifications

This part 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 are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for non-terminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
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 easily added.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 below is written following this style, as are the examples in this section (where space permits). The central problem 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  ;
```

match this algorithm. Rules such as

```
list   :   item
       |   list  ','  item
       ;
```

and

```
seq    :   item
       |   seq  item
       ;
```

frequently arise when writing specifications of sequences and lists. 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 is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if an extremely long sequence is read (although **yacc** can process very large stacks). Thus, you should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq   :   /* empty */
      |   seq  item
      ;
```

using an empty rule. 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 these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
   int dflag;
%}
  ...   other declarations ...

%%

prog   :   decls  stats
       ;

decls  :   /* empty */
       {
              dflag = 1;
       }
       |   decls  declaration
       ;

stats  :   /* empty */
       {
              dflag = 0;
       }
       |   stats  statement
       ;

    other rules
```

specifies a program that consists of zero or more declarations followed by zero or more statements. 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 back-door 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 you to use words like `if`, which are normally reserved as label or variable names, provided that such use does not conflict with the valid 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. You can make a stab at it using the mechanism described in the last subsection, but it is difficult.

Advanced Topics

This part 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`. The `YYACCEPT` macro causes `yyparse()` to return the value 0; `YYERROR` causes the parser to behave as if the current input symbol had been a syntax error; `yyerror()` is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers 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, `$` followed by a digit.

```
sent   :   adj  noun  verb  adj  noun
       {
           look at the sentence ...
       }
       ;
adj    :   THE
       {
                $$ = THE;
       }
       |   YOUNG
       {
                $$ = YOUNG;
       }
       ...
       ;
noun   :   DOG
       {
           $$ = DOG;
       }
       |   CRONE
       {
           if( $0 == YOUNG )
           {
               (void) printf( "what?\n" );
           }
           $$ = CRONE;
       }
       ;
       ...
```

In this case, the digit may be 0 or negative. 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. Nevertheless, at times this mechanism prevents 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 is strictly type checked. The **`yacc`** value stack is declared to be a `union` of the various types of values desired. You declare the union and associate union member names with each token and non-terminal symbol having a value. When the value is referenced through a `$$` or `$`*n* construction, **`yacc`** will automatically insert the appropriate union name so that no unwanted conversions take place.

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 user since other subroutines, 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 non-terminals. Finally, there is a mechanism for describing the type of those few values where **`yacc`** cannot easily determine the type.

To declare the union, you include

```
%union
{
    body of union
}
```

in the declaration section. 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 into the **y.tab.h** file as `YYSTYPE`.

Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and non-terminal 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>  '+'  '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name `optype`. Another keyword, `%type`, is used to associate union member names with non-terminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

to associate the union member `nodetype` with the non-terminal symbols `expr` and `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 `$0`) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between `<` and `>` immediately after the first `$`. The example below

```
rule   :   aaa
                {
                   $<intval>$ = 3;
                }
                bbb
       {
           fun( $<intval>2, $<other>0 );
       }
       ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2 below. 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.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies and so forth are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, 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 decides whether the next token (skipping blanks, new-lines, comments, and so on) is a colon. If so, it returns the token `C_IDENTIFIER`. Otherwise, it returns `IDENTIFIER`. Literals (quoted strings) are also returned as `IDENTIFIER`s but never as part of `C_IDENTIFIER`s.

```
    /* grammar for the input to yacc */

    /* basic entries */
%token     IDENTIFIER    /* includes identifiers and literals */
%token     C_IDENTIFIER  /* identifier (but not literal) followed by a : */
%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 */

%token    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
       |   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 invalid with % type */
       |  IDENTIFIER NUMBER   /* Note: invalid with % type */
       ;

   /* rule 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 ';'
       ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled `a` through `z` and accepts arithmetic expressions made up of the operators `+, -, *, /, %, &, |`, and the assignment operators.

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 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 precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer 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 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
          {
             (void) 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
          {
                $$ = $1 / $3;
          }
          |   exp '%' expr
          {
                 $$ = $1 % $3;
          }
          |   expr '&' expr
          {
                $$ = $1 & $3;
          }
          |   expr '|' expr
          {
                $$ = $1 | $3;
          }
          |  '-' expr  %prec UMINUS
          {
                $$ = -$2;
          }
          |  LETTER
          {
                $$ = reg[$1];
          }
          |  number
          ;

number    :   DIGIT
          {
                 $$ = $1; base = ($1==0) ? 8 ; 10;
          }
          |   number DIGIT
          {
                 $$ = base * $1 + $2;
          }
          ;

%%      /* beginning of subroutines section */

int yylex( )   /* lexical analysis routine */
{              /* return LETTER for lowercase letter, */
               /* yylval = 0 through 25 */
               /* returns DIGIT for digit, yylval = 0 through 9 */
               /* all other characters are returned immediately */
          int c;
                      /*skip blanks*/
          while ((c = getchar()) == ' ')
                 ;

                      /* c is now nonblank */

          if (islower(c))
          {
                     yylval = c - 'a';
                     return (LETTER);
          }
          if (isdigit(c))
          }
                     yylval = c - '0';
                     return (DIGIT);

          }
          return (c);
}
```

2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator in Example 1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, and the arithmetic operations `+`, `-`, `*`, `/`, and unary `-`. It uses the registers `a` through `z`. Moreover, it 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 Example 1; 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 the entire strategy depends strongly on being able to assign structures and unions in C language. 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. 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 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 value expression in the second example, but this fact is not known until the comma 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 conflict 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 instructive. 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, 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 language 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 invalid 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 '*' '/'

%%      /* beginning of rules section */

lines   :   /* empty */
   |    lines line
   ;
line   :   dexp '\n'
   {
      (void)printf("%15.8f\n", $1);
   }
   |   vexp '\n'
```

```
    {
       (void)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
    {
       $$ = $1 / $3;
    }
    |   '-' dexp
    {
       $$ = -$2;
    }
    |   '(' dexp ')'
    {
       $$ = $2;
    }
    ;
vexp    :   dexp
    {
       $$.hi = $$.lo = $1;
    }
    |   '(' dexp ',' dexp ')'
    {
       $$.lo = $2;
       $$.hi = $4;
       if($$.lo > $$.hi)
       {
          (void) printf("interval out of order\n");
          YYERROR;
       }
    }
    |   VREG
    {
       $$ = 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
    {
       $$.hi = -$2.lo; $$.lo = -$2.hi;
    }
    |   '(' vexp ')'
    {
       $$ = $2;
    }
    ;

 %%   /* beginning of subroutines section */

 # define BSZ 50   /* buffer size for floating point number */

    /* lexical analysis */

 int yylex()
 {
    register int c;

       /* skip over blanks */

    while ((c=getchar()) == ' ')
       ;
    if (isupper(c))
    {
       yylval.ival = c - 'A';
       return(VREG);
    }
    if (islower(c))
    {
       yylval.ival = c - 'a';
       return(DREG);
    }

       /* gobble up digits, points, exponents */

    if (isdigit(c) || c == '.')
    {
       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)
          (void)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 vmul, vdiv 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.)
   {
      (void) 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));
}
```

2
Analysis

Replace with Part 2 tab

Part 2 - Analysis

Part 2 Analysis

8
Introduction to Analysis

8 Introduction to Analysis

Introduction

By using tools to analyze source files and executables you can:

- Locate and correct problems
- Obtain statistics on usage and performance timings
- Improve program reliability and performance

Many analysis tools exist. See "Concurrent Computer Corporation Compilation Systems" section in Chapter 1 for an extensive list of these and other utilities.

Although not discussed in this manual, the C beautifier, **`cb(1)`**, can assist in analysis; it makes C source files more readable with judicious placement of spaces and indentation. The **`xref(1)`** utility combines many cross referencing aspects of **`cscope`** and inconsistency-detecting aspects of **`lint`** for Fortran source files. See the man page for details.

This part of the manual discusses the analysis of source files and executables.

Chapter 9 ("Browsing Through Your Code with cscope") discusses cross referencing, searching, and editing C, **`lex`**, and **`yacc`** source files with **`cscope`**.

Chapter 10 ("Analyzing Your Code with lint") describes using **`lint`** on C source files to flag inconsistent use, non-portable code, and suspicious constructs.

Chapter 11 ("Performance Analysis") explains how to use **`analyze`** to optimize programs or obtain performance profiles on programs and **`report`** to generate reports from **`analyze`**'s output.

9
Browsing Through Your Code with cscope

9
Browsing Through Your Code with cscope

Introduction

The **cscope** browser is an interactive program that locates specified elements of code in C, **lex**, or **yacc** source files. It lets you search and edit your source files more efficiently than you could with a typical editor. **cscope** has this capability because it can identify function calls and C language identifiers and keywords. This chapter contains a tutorial on the **cscope** browser.

How cscope Works

When you invoke **cscope** for a set of C, **lex**, or **yacc** source files, it builds a symbol cross-reference table for the functions, function calls, macros, variables, and preprocessor symbols in those files. It then lets you query that table about the locations of symbols you specify. First, it presents a menu and asks you to choose the type of search you would like to have performed. You may, for instance, want **cscope** to find all functions that call a specified function.

When **cscope** has completed this search, it prints a list. Each list entry contains the name of the file, the number of the line, and the text of the line in which **cscope** has found the specified code. In this example, the list will also include the names of the functions that call the specified function. If you choose the latter, **cscope** invokes the editor for the file in which the line appears, with the cursor on that line. You may now view the code in context and edit the file as you would any other file. You can then return to the menu from the editor to request a new search.

Because of the procedure you follow there is no single set of instructions for using **cscope**. For an extended example of its use, review the **cscope** session described in the next section. It shows how you can locate a bug in a program without learning all the code.

How to Use cscope

In the first example, an error message, `out of storage`, appears intermittently in the program **prog**, just as the program starts up. The following series of steps shows you how to use **cscope** to locate the parts of the code that are generating the message.

Step 1: Set Up the Environment

cscope is a screen-oriented tool that can only be used on terminals listed in the Terminal Information Utilities (**terminfo**) database. Be sure you have set the `TERM` environment variable to your terminal type so that **cscope** can verify that it is listed in the **terminfo** database. If you have not done so, assign a value to `TERM` and export it to the shell as follows:

TERM=*term_name* **export TERM**

You may now want to assign a value to the `EDITOR` environment variable. By default, **cscope** invokes the **vi** editor. (The examples in this chapter illustrate **vi** usage.) If you prefer not to use **vi**, set the `EDITOR` environment variable to the editor of your choice and export `EDITOR:`

```
EDITOR=emacs export EDITOR
```

Note that you may have to write an interface between **cscope** and your editor. For details, see “Command Line Syntax for Editors” on page 9-18.

If you want to use **cscope** only for browsing (without editing), you can set the `VIEWER` environment variable to **pg** and export `VIEWER.` **cscope** will then invoke **pg** instead of **vi**.

An environment variable called `VPATH` can be set to specify directories to be searched for source files. See “Using Viewpaths” on page 9-13.

Step 2: Invoke cscope

By default, **cscope** builds a symbol cross-reference table for all the C, **lex**, and **yacc** source files in the current directory, and for any included header files in the current directory or the standard place. If all the source files for the program to be browsed are in the current directory, and if its header files are there or in the standard place, invoke **cscope** without arguments:

```
cscope
```

To browse through selected source files, invoke **cscope** with the names of those files as arguments:

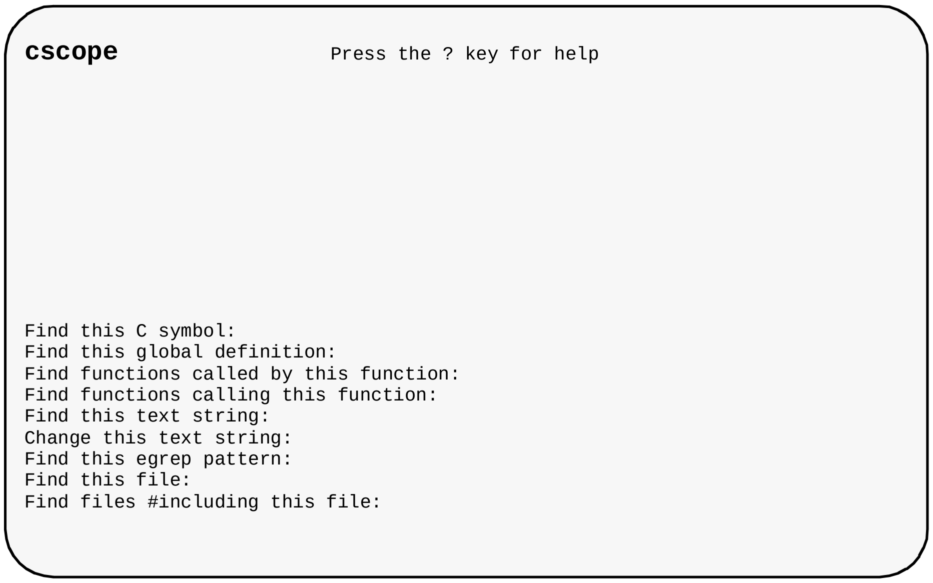
```
cscope file1.c file2.c file3.h
```

For other ways to invoke **cscope**, see “Command Line Options” on page 9-10.

cscope builds the symbol cross-reference table the first time it is used on the source files for the program to be browsed. By default, the table is stored in the file **cscope.out** in the current directory. On a subsequent invocation, **cscope** rebuilds the cross-reference only if a source file has been modified or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files are copied from the old cross-reference, which makes rebuilding faster than the initial build and startup time less for subsequent invocations.

Step 3: Locate the Code

Now you can begin to identify the problem that is causing the error message `out of storage` to be printed. You have invoked **`cscope`**, and the cross-reference table has been built. The **`cscope`** menu of tasks appears on the screen:



```
cscope                    Press the ? key for help









Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-1. The cscope Menu of Tasks

Press the RETURN or Enter key to move the cursor down the screen (with wraparound at the bottom of the display), and Ctrl-p to move the cursor up; or use the up arrow and down arrow keys if your keyboard has them. You can manipulate the menu, and perform other tasks, with the following single-key commands:

Table 9-1. Menu Manipulation Commands

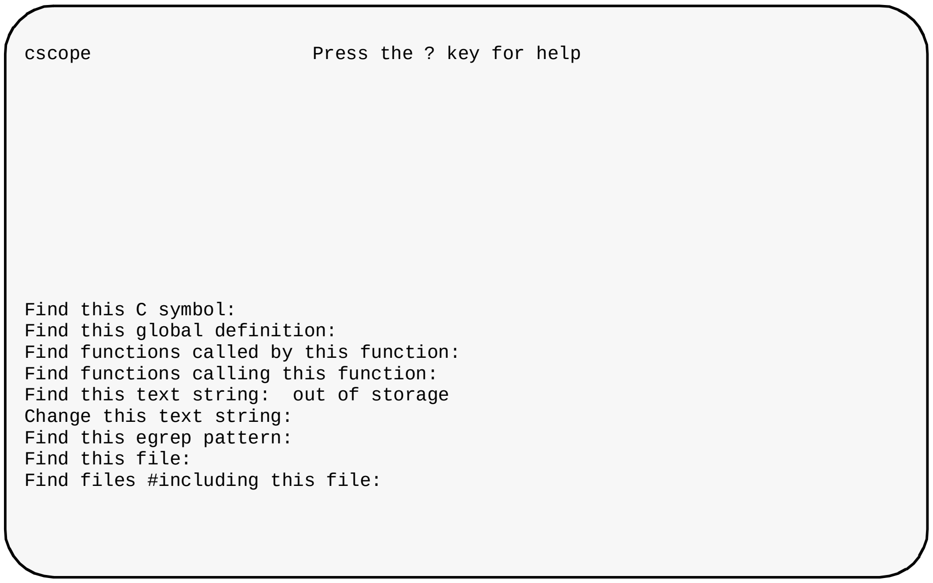
TAB	move to next input field
RETURN	move to next input field
Ctrl-n	move to next input field
Ctrl-p	move to previous input field
Ctrl-y	search with the last pattern typed
Ctrl-b	move to previous input field and search pattern
Ctrl-f	recall next input field and search pattern
Ctrl-c	toggle ignore/use letter case when searching (a search for `FILE` will match, for example, `File` and `file` when ignoring letter case)
Ctrl-r	rebuild the cross-reference
!	start an interactive shell (type Ctrl-d to return to **`cscope`**)

Table 9-1. Menu Manipulation Commands (Cont.)

Ctrl-l	redraw the screen
?	display list of commands
Ctrl-d	exit **cscope**

If the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash (\) before the character.

Now move the cursor to the fifth menu item, `Find this text string`, enter the text `out of storage`, and press the RETURN key:



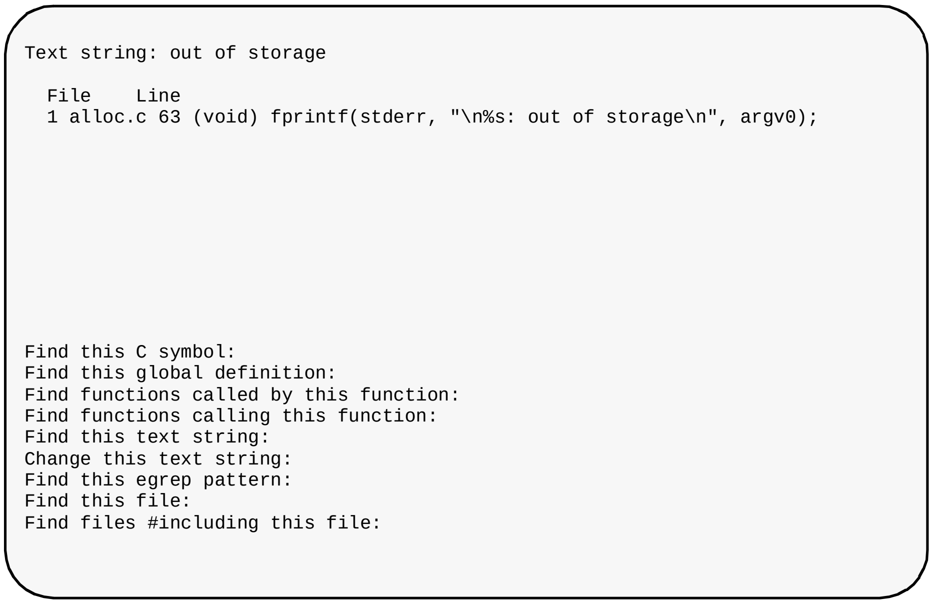
```
cscope                    Press the ? key for help










Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:  out of storage
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-2. Requesting a Search for a Text String

NOTE

Follow the same procedure to perform any other task listed in the menu except the sixth, `Change this text string`. Because this task is slightly more complex than the others, there is a different procedure for performing it. For a description of how to change a text string, see “Examples” on page 9-14.

cscope searches for the specified text, finds one line that contains it, and reports its finding as follows:



```
Text string: out of storage

  File    Line
  1 alloc.c 63 (void) fprintf(stderr, "\n%s: out of storage\n", argv0);









Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-3. cscope Lists Lines Containing the Text String

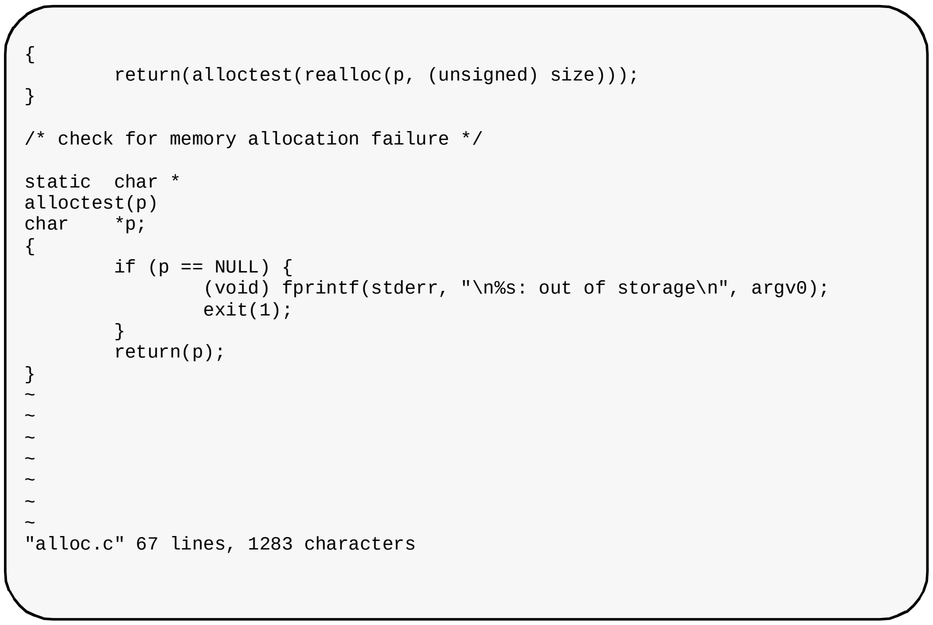
After **cscope** shows you the results of a successful search, you have several options. You may want to change the lines or examine the code surrounding it in the editor. Or, if **cscope** has found so many lines that a list of them will not fit on the screen at once, you may want to look at the next part of the list. You can even filter out unwanted lines from the list **cscope** has found. The following table shows the commands available after **cscope** has found the specified text:

Table 9-2. Commands for Use after Initial Search

1-9	edit the file referenced by this line (the number you type corresponds to an item in the list of lines printed by **cscope**)
space bar	display next set of matching lines
+	display next set of matching lines
Ctrl-v	display next set of matching lines
-	display previous set of matching lines
Ctrl-e	edit displayed files in order
>	write the list of lines being displayed to a file
>>	append the list of lines being displayed to a file
<	read lines from a file
^	filter all lines through a shell command, replacing the lines originally found with the output of the shell command
|	pipe all lines to a shell command, displaying the output of the shell command without changing the list of lines found

If the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash before the character.

Now examine the code around the newly found line. Enter `1` (the number of the line in the list). The editor will be invoked with the file **`alloc.c`**; the cursor will be at the beginning of line 63 of **`alloc.c`**:

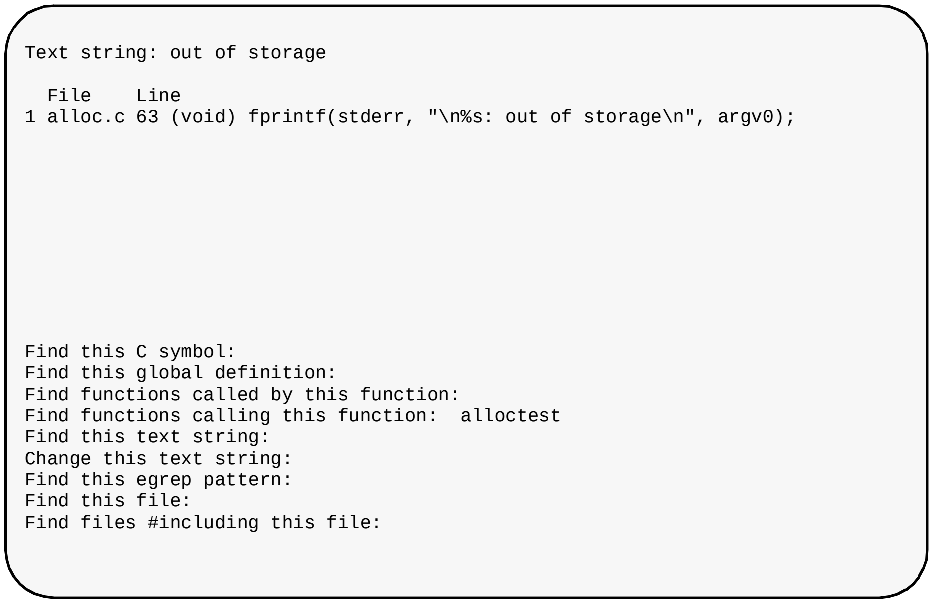


```
{
        return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static  char *
alloctest(p)
char    *p;
{
        if (p == NULL) {
                (void) fprintf(stderr, "\n%s: out of storage\n", argv0);
                exit(1);
        }
        return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

Screen 9-4. Examining a Line of Code Found by cscope

You can see that the error message is generated when the variable `p` is `NULL.`To determine how an argument passed to `alloctest()` could have been `NULL,` you must first identify the functions that call `alloctest()`.

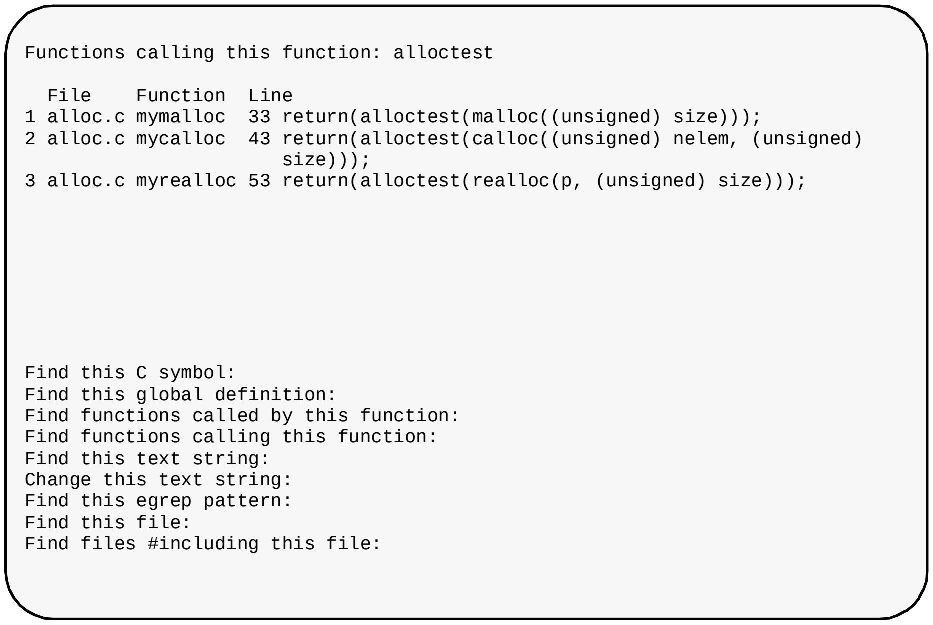
Exit the editor by using normal quit conventions. You are returned to the menu of tasks. Now type `alloctest` after the fourth item, `Find functions calling this function`:



```
Text string: out of storage

  File    Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s: out of storage\n", argv0);









Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-5. Requesting a List of Functions That Call alloctest()

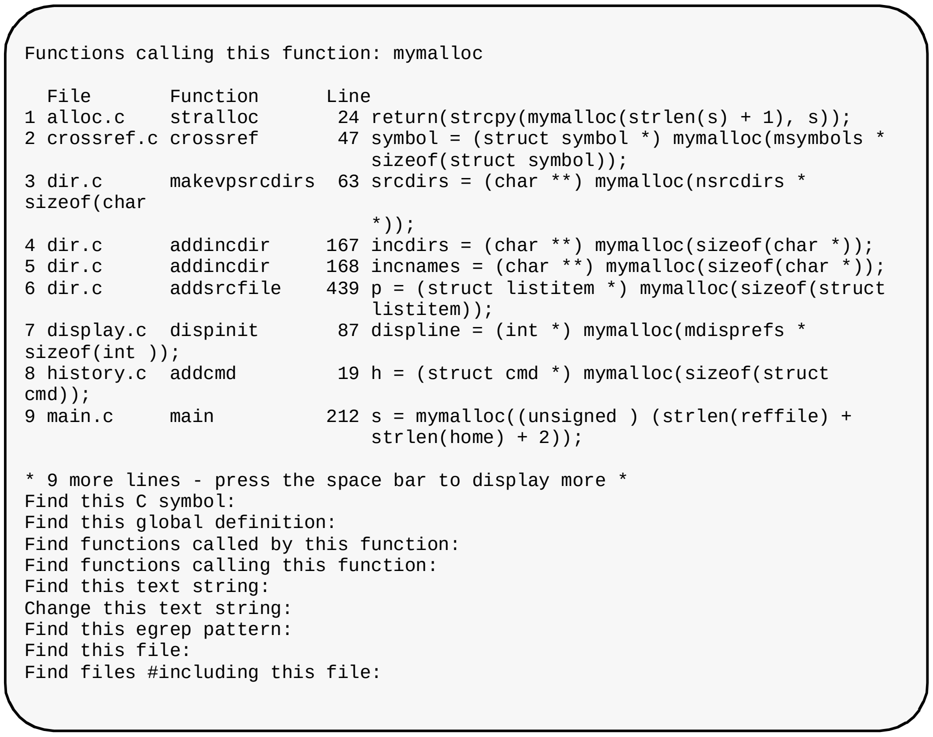
`cscope` finds and lists three such functions:



```
Functions calling this function: alloctest

  File    Function  Line
1 alloc.c mymalloc  33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc  43 return(alloctest(calloc((unsigned) nelem, (unsigned)
                       size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));







Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-6. cscope Lists Functions That Call alloctest()

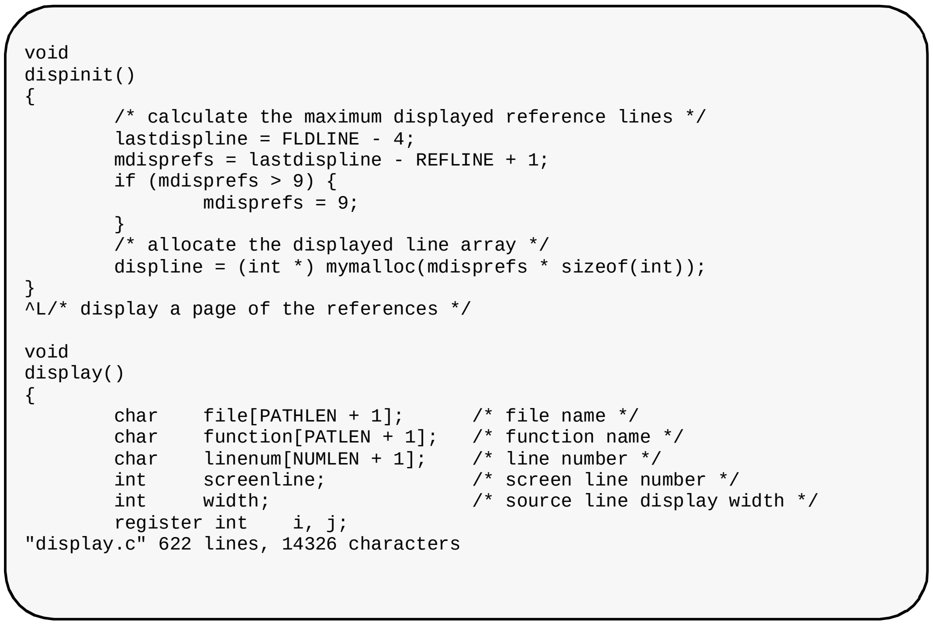
Now you want to know which functions call `mymalloc()`. **`cscope`** finds ten such functions. It lists nine of them on the screen and instructs you to press the space bar to see the rest of the list:



```
Functions calling this function: mymalloc

  File       Function      Line
1 alloc.c    stralloc       24 return(strcpy(mymalloc(strlen(s) + 1), s));
2 crossref.c crossref       47 symbol = (struct symbol *) mymalloc(msymbols *
                               sizeof(struct symbol));
3 dir.c      makevpsrcdirs  63 srcdirs = (char **) mymalloc(nsrcdirs *
sizeof(char
                               *));
4 dir.c      addincdir     167 incdirs = (char **) mymalloc(sizeof(char *));
5 dir.c      addincdir     168 incnames = (char **) mymalloc(sizeof(char *));
6 dir.c      addsrcfile    439 p = (struct listitem *) mymalloc(sizeof(struct
                               listitem));
7 display.c  dispinit       87 displine = (int *) mymalloc(mdisprefs *
sizeof(int ));
8 history.c  addcmd         19 h = (struct cmd *) mymalloc(sizeof(struct
cmd));
9 main.c     main          212 s = mymalloc((unsigned ) (strlen(reffile) +
                               strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Screen 9-7. cscope Lists Functions That Call mymalloc()

Because you know that the error message `out of storage` is generated at the beginning of the program, you can guess that the problem may have occurred in the function `dispinit()` (display initialization). To view `dispinit()`, the seventh function on the list, type 7:



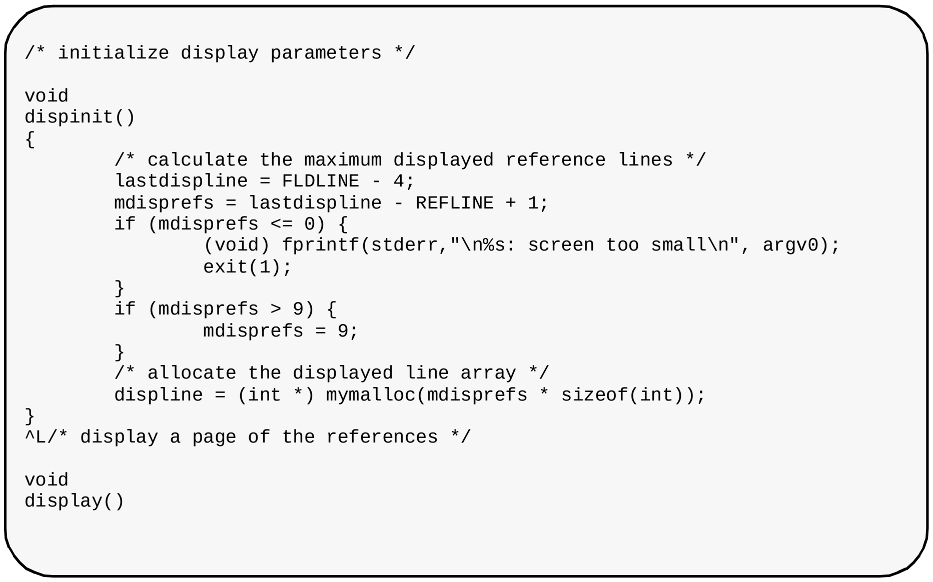
```
void
dispinit()
{
        /* calculate the maximum displayed reference lines */
        lastdispline = FLDLINE - 4;
        mdisprefs = lastdispline - REFLINE + 1;
        if (mdisprefs > 9) {
                mdisprefs = 9;
        }
        /* allocate the displayed line array */
        displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
        char    file[PATHLEN + 1];      /* file name */
        char    function[PATLEN + 1];   /* function name */
        char    linenum[NUMLEN + 1];    /* line number */
        int     screenline;             /* screen line number */
        int     width;                  /* source line display width */
        register int    i, j;
"display.c" 622 lines, 14326 characters
```

Screen 9-8. Viewing dispinit() in the Editor

`mymalloc()` failed because it was called either with a very large number or a negative number. By examining the possible values of `FLDLINE` and `REFLINE`, you can see that there are scenarios in which the value of `mdisprefs` is negative, for example, when you are trying to call `mymalloc()` with a negative number.

Step 4: Edit the Code

On a windowing terminal you may have multiple windows of arbitrary size. The error message `out of storage` might have appeared as a result of running **`prog`**. That may have been one of the situations in which `mymalloc()` was called with a negative number. Now you want to be sure that when the program aborts in this scenario in the future, it does so after printing the more significant error message `screen too small`. Edit the function `dispinit()` as follows:



```
/* initialize display parameters */

void
dispinit()
{
        /* calculate the maximum displayed reference lines */
        lastdispline = FLDLINE - 4;
        mdisprefs = lastdispline - REFLINE + 1;
        if (mdisprefs <= 0) {
                (void) fprintf(stderr,"\n%s: screen too small\n", argv0);
                exit(1);
        }
        if (mdisprefs > 9) {
                mdisprefs = 9;
        }
        /* allocate the displayed line array */
        displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
```

Screen 9-9. Using cscope to Fix the Problem

You have fixed the problem that you began investigating at the beginning of this section. Now if **`prog`** is run in a window with too few lines, it will not simply fail with the vague error message `out of storage`. Instead, it will check the window size and generate a more significant error message before exiting.

Command Line Options

As noted, **`cscope`** builds a symbol cross-reference table for the C, **`lex`**, and **`yacc`** source files in the current directory by default.

`cscope`

is equivalent to

`cscope *.[chly]`

The following example shows how you can browse through selected source files by invoking **`cscope`** with the names of those files as arguments:

`cscope file1.c file2.c file3.h`

`cscope` provides command line options that allow you greater flexibility in specifying source files to be included in the cross-reference. When you invoke **`cscope`** with the **`-s`** option and any number of directory names (separated by commas)

`cscope -s` *dir,dir,dir*

`cscope` will build a cross-reference for all the source files in the specified directories as well as the current directory. To browse through all of the source files whose names are listed in *file* (file names separated by spaces, tabs, or new-lines), invoke **cscope** with the **-i** option and the name of the file containing the list:

```
cscope -i file
```

If your source files are in a directory tree, the following commands will allow you to browse through all of them easily:

```
find . -name '*.[chly]' -print | sort > file
cscope  -i  file
```

Note that if this option is selected, **cscope** ignores any other files appearing on the command line.

The **-I** option to **cscope** is similar to the **-I** option to **cc**. By default, **cscope** searches for included header files in the current directory, then the standard place. If you want **cscope** to search for an included header file in a different directory, specify the path of the directory with **-I**:

```
cscope -I dir
```

In this example, **cscope** will search the directory *dir* for `#include` files called into the source files in the current directory. Directories are searched for `#include` files in the following order:

1. the current directory;
2. the directories specified with **-I**;
3. the standard place for header files, usually **/usr/include**.

You can invoke the **-I** option more than once on a command line. **cscope** will search the specified directories in the order they appear on the command line.

You can specify a cross-reference file other than the default **cscope.out** by invoking the **-f** option. This is useful for keeping separate symbol cross-reference files in the same directory. You may want to do this if two programs are in the same directory, but do not share all the same files:

```
cscope -f admin.ref admin.c common.c aux.c libs.c
cscope -f delta.ref delta.c common.c aux.c libs.c
```

In this example, the source files for two programs, **admin** and **delta**, are in the same directory, but the programs consist of different groups of files. By specifying different symbol cross-reference files when you invoke **cscope** for each set of source files, the cross-reference information for the two programs is kept separate.

You can use the **-p***n* option to specify that **cscope** display the path name, or part of the path name, of a file when it lists the results of a search. The number you give to **-p** stands for the last *n* elements of the path name you want to be displayed. The default is `1`, the name of the file itself. So if your current directory is **home/common**, the command

```
cscope -p2
```

will cause **cscope** to display **common/file1.c**, **common/file2.c**, and so forth when it lists the results of a search.

If the program you want to browse contains a large number of source files, you can use the **`-b`** option to tell **`cscope`** to stop after it has built a cross-reference; **`cscope`** will not display a menu of tasks. When you use **`cscope -b`** in a pipeline with the **`batch`** command, **`cscope`** will build the cross-reference in the background:

```
echo 'cscope -b' | batch
```

NOTE

> See **`batch(1)`** for more information.

Once the cross-reference is built (and as long as you have not changed a source file or the list of source files in the meantime), you need only specify

```
cscope
```

for the cross-reference to be copied and the menu of tasks to be displayed in the normal way. In other words, you can use this sequence of commands when you want to continue working without having to wait for **`cscope`** to finish its initial processing.

The **`-d`** option instructs **`cscope`** not to update the symbol cross-reference. You can use it to save time — **`cscope`** will not check the source files for changes — if you are sure that no such changes have been made.

NOTE

> Use the **`-d`** option with care. If you specify **`-d`** under the erroneous impression that your source files have not been changed, **`cscope`** will refer to an outdated symbol cross-reference in responding to your queries.

To use **`cscope`** separately on several programs in the same directory structure while keeping the databases in the same directory, use the **`-f`** and **`-i`** options to rename the **`cscope.out`** and **`cscope.files`** file as follows:

```
find dir1 -name '*.[chlyCGHL}' -print >dir1.files
find dir2 -name '*.[chlyCGHL}' -print >dir2.files
cscope -b -f dir1.db -i dir1.files
cscope -b -f dir2.db -i dir2.files
```

Call **`cscope`** with:

```
cscope -d -f dir2.db
```

Options used only when building the database, such as **`-i`** are not needed with the **`-d`** option. Use the **`-P`** option to give the path to relative file names so the script does not have to change to the directory where the database was built.

The **`-F`***file* option reads symbol reference lines from *file*, similar to the **<** command.

The **`-q`** option builds an inverted index for quick symbol searching. If you use this option with the **`-f`** option, you must use **`-f`** on every call to **`cscope`** including building the database, because it changes the names of the inverted index files. For large databases, you will be able to find a symbol in a few seconds instead of the several minutes it can take to build without **`-q`**, at the expense of about twice as much database disk space and build CPU time. Updating a **`-q`** database takes about half as long as building it. It contains binary numbers, so it is portable only between machines with the same byte ordering.

The **`-q`** option makes it practical to have databases for entire projects. If you try to build a project database and get a `file too large` message, you need to get your login's `ulimit` raised by your system administrator. (See **`sh(1)`** for information on the shell built-in **`ulimit`** command.) If you get the `no space left on device` message, you will have to use a file system with more space. You can change the temporary file system by setting the `TMPDIR` environment variable. If you have enough space to build the database but not to rebuild it after some files have changed, try removing the inverted index **`cscope.in.out`** and **`cscope.po.out`** files. If you still don't have enough space to rebuild, remove the **`cscope.out`** file.

Check the **`cscope(1)`** page for other command line options.

Using Viewpaths

`cscope` searches for source files in the current directory by default. When the environment variable `VPATH` is set, **`cscope`** searches for source files in directories that comprise your viewpath. A viewpath is an ordered list of directories, each of which has the same directory structure below it.

For example, suppose you are part of a software project. There is an “official” set of source files in directories below **`/fs1/ofc`**. Each user has a home directory (**`/usr/you`**). If you make changes to the software system, you may have copies of just those files you are changing in **`/usr/you/src/cmd/prog1`**. The official versions of the entire program can be found in the directory **`/fs1/ofc/src/cmd/prog1`**.

Suppose you use **`cscope`** to browse through the three files that comprise **`prog1`**, namely, **`f1.c`**, **`f2.c,`** and **`f3.c.`** You would set `VPATH` to **`/usr/you`** and **`/fs1/ofc`** and export it, as in

```
VPATH=/usr/you:/fs1/ofc export VPATH
```

You would then make your current directory **`/usr/you/src/cmd/prog1`**, and invoke **`cscope`**:

```
cscope
```

The program will locate all files in the viewpath. In case duplicates are found, **`cscope`** uses the file whose parent directory appears earlier in `VPATH`. Thus if **`f2.c`** is in your directory (and all three files are in the official directory), **`cscope`** will examine **`f2.c`** from your directory and **`f1.c`** and **`f3.c`** from the official directory.

The first directory in `VPATH` must be a prefix (usually `$HOME`) of the directory you will be working in. Each colon-separated directory in `VPATH` must be absolute: it should begin at **`/`**.

Stacking cscope and Editor Calls

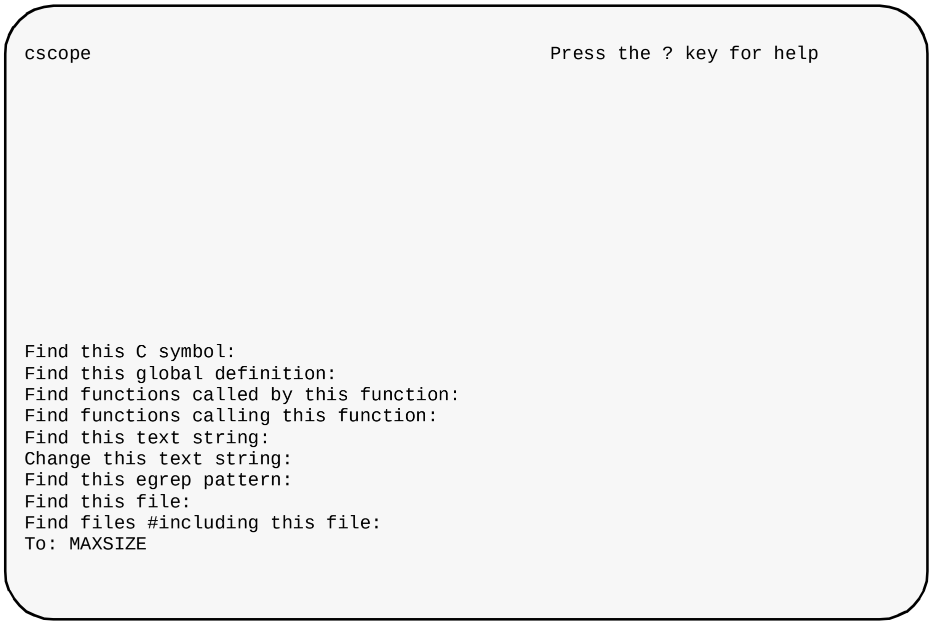
cscope and editor calls can be stacked. That means that when **cscope** puts you in the editor to view a reference to a symbol and there is another reference of interest, you can invoke **cscope** again from within the editor to view the second reference without exiting the current invocation of either **cscope** or the editor. You can then back up by exiting the most recent invocation with the appropriate **cscope** and editor commands.

Examples

This section presents examples of how **cscope** can be used to perform three tasks: changing a constant to a preprocessor symbol, adding an argument to a function, and changing the value of a variable. The first example demonstrates the procedure for changing a text string, which differs slightly from the other tasks on the **cscope** menu. Once you have entered the text string to be changed, **cscope** prompts you for the new text, displays the lines containing the old text, and waits for you to specify which of these lines you want it to change.

Changing a Constant to a Preprocessor Symbol

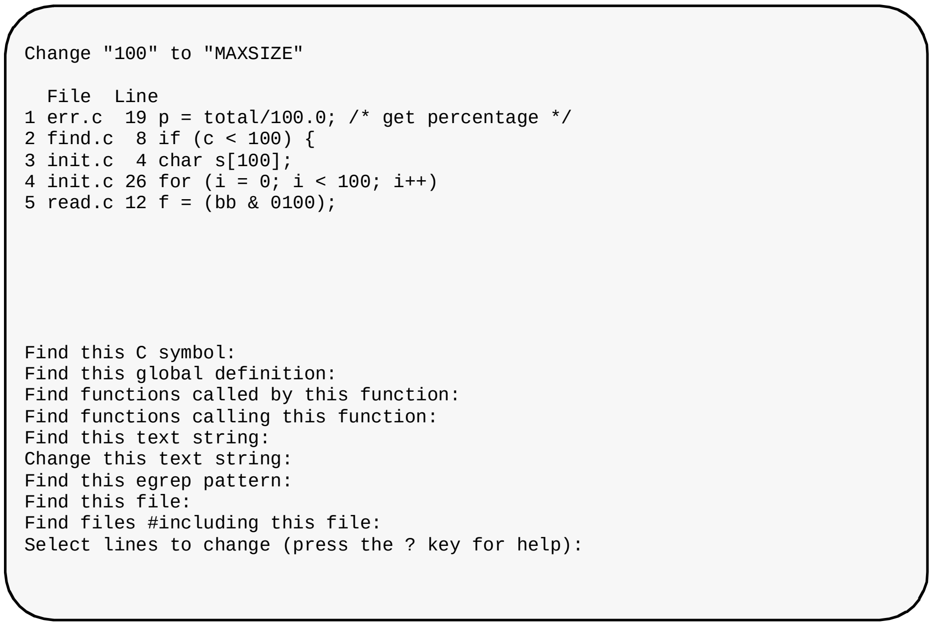
Suppose you want to change a constant, `100`, to a preprocessor symbol, `MAXSIZE.` Select the sixth menu item, `Change this text string`, and enter `\100`. The `1` must be escaped with a backslash because it has a special meaning (item 1 on the menu) to **cscope**. Press RETURN. **cscope** will prompt you for the new text string. Type `MAXSIZE`:



```
cscope                                               Press the ? key for help











Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
To: MAXSIZE
```

Screen 9-10. Changing a Text String

cscope displays the lines containing the specified text string, and waits for you to select those in which you want the text to be changed:



```
Change "100" to "MAXSIZE"

  File  Line
1 err.c  19 p = total/100.0; /* get percentage */
2 find.c  8 if (c < 100) {
3 init.c  4 char s[100];
4 init.c 26 for (i = 0; i < 100; i++)
5 read.c 12 f = (bb & 0100);







Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

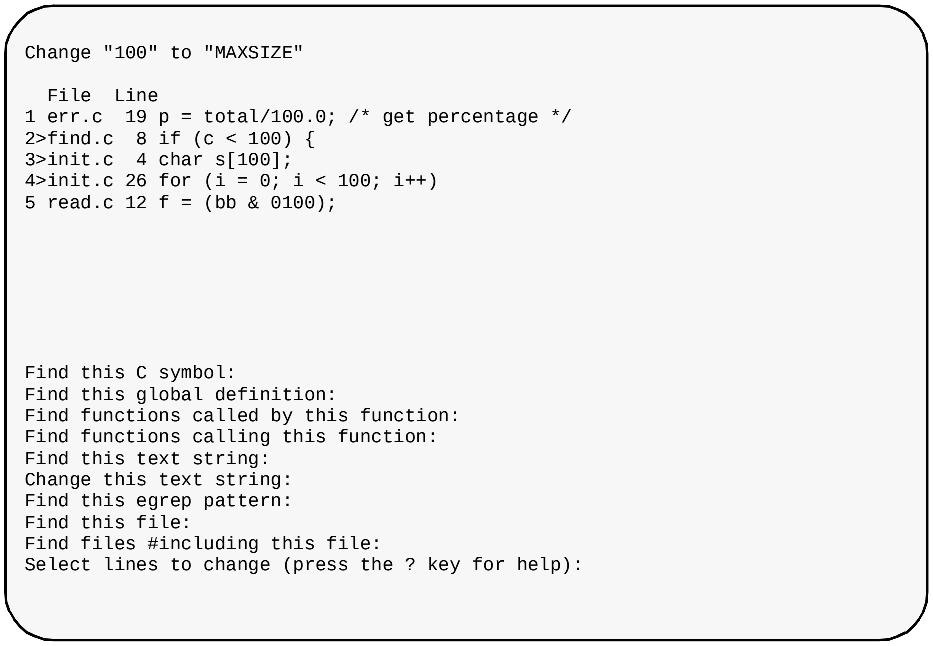
Screen 9-11. cscope Prompts for Lines to Be Changed

You know that the constant `100` in lines 2, 3, and 4 of the list (lines 4, 26, and 8 of the listed source files) should be changed to `MAXSIZE.` You also know that `100` in **`err.c`** and `0100.0` in **`read.c`** (lines 1 and 5 of the list) should not be changed. You select the lines you want changed with the following single-key commands:

Table 9-3. Commands for Selecting Lines to Be Changed

`1-9`	mark or "unmark" the line to be changed
`*`	mark or "unmark" all displayed lines to be changed
`space bar`	display next set of lines
`+`	display next set of lines
`-`	display previous set of lines
`a`	mark or "unmark" all lines to be changed
`Ctrl-d`	change the marked lines and exit
`ESC`	exit without changing the marked lines

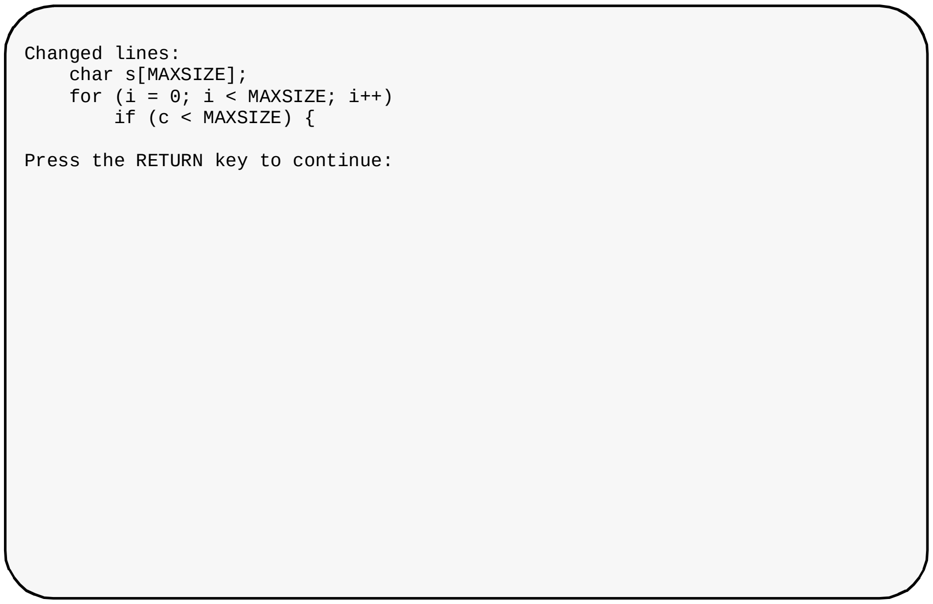
In this case, enter `2`, `3`, and `4`. Note that the numbers you type are not printed on the screen. Instead, **cscope** marks each list item you want to be changed by printing a `>` (greater than) symbol after its line number in the list:



```
Change "100" to "MAXSIZE"

  File  Line
1 err.c  19 p = total/100.0; /* get percentage */
2>find.c  8 if (c < 100) {
3>init.c  4 char s[100];
4>init.c 26 for (i = 0; i < 100; i++)
5 read.c 12 f = (bb & 0100);








Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

Screen 9-12. Marking Lines to Be Changed

Now press Ctrl-d to change the selected lines. **cscope** displays the lines that have been changed and prompts you to continue:

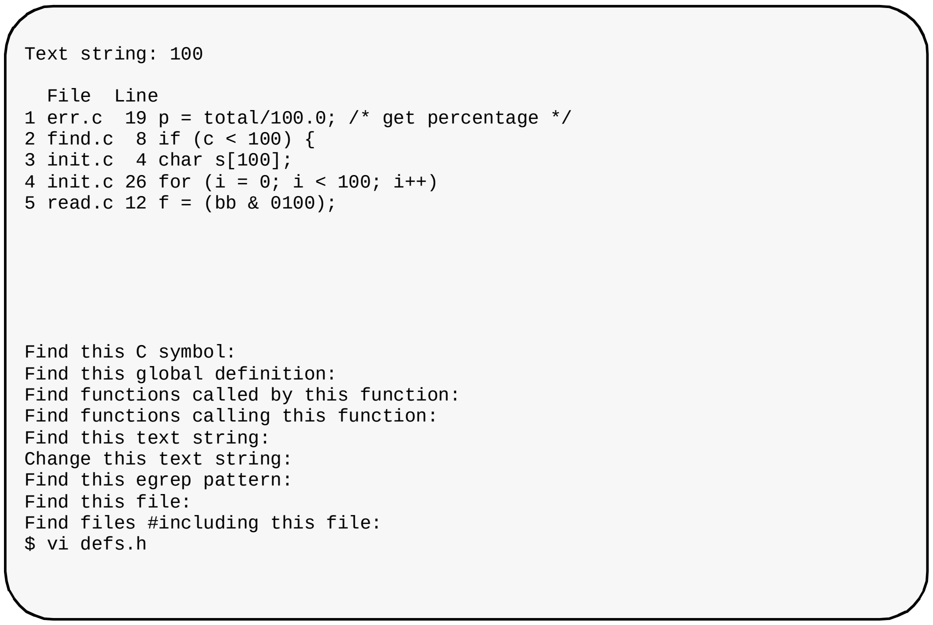


```
Changed lines:
    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
        if (c < MAXSIZE) {

Press the RETURN key to continue:
```

Screen 9-13. cscope Displays Changed Lines of Text

When you press `RETURN` in response to this prompt, **`cscope`** redraws the screen, restoring it to its state before you selected the lines to be changed, as shown in the screen below.

The next step is to add the `#define` for the new symbol `MAXSIZE.` Because the header file in which the `#define` is to appear is not among the files whose lines are displayed, you must escape to the shell by typing `!`. The shell prompt will appear at the bottom of the screen. Then enter the editor and add the `#define`:



```
Text string: 100

  File  Line
1 err.c  19 p = total/100.0; /* get percentage */
2 find.c  8 if (c < 100) {
3 init.c  4 char s[100];
4 init.c 26 for (i = 0; i < 100; i++)
5 read.c 12 f = (bb & 0100);







Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

Screen 9-14. Escaping from cscope to the Shell

To resume the **`cscope`** session, quit the editor and press `Ctrl-d` to exit the shell.

Adding an Argument to a Function

Adding an argument to a function involves two steps: editing the function itself and adding the new argument to every place in the code where the function is called. **`cscope`** makes that easy.

First, edit the function by using the second menu item, `Find this global definition`. Next, find out where the function is called. Use the fourth menu item, `Find functions calling this function`, to get a list of all the functions that call it. With this list, you can either invoke the editor for each line found by entering the list number of the line individually, or invoke the editor for all the lines automatically by pressing `Ctrl-e`. Using **`cscope`** to make this type of change assures that none of the functions you need to edit will be overlooked.

Changing the Value of a Variable

The value of **`cscope`** as a browser becomes apparent when you want to see how a proposed change will affect your code. If you want to change the value of a variable or preprocessor symbol, use the first menu item, `Find this C symbol`, to obtain a list of references that will be affected. Then use the editor to examine each one. This will help you predict the overall effects of your proposed change. You can also use this menu to verify that your changes have been made.

Technical Tips

This section describes certain problems that may arise when you use **`cscope`** and how to avoid them.

Unknown Terminal Type

You may see the error message

```
Sorry, I don't know anything about your "term" terminal
```

If this message appears, your terminal may not be listed in the Terminal Information Utilities (**`terminfo`**) database that is currently loaded. Make sure you have assigned the correct value to `TERM.` If the message reappears, try reloading the Terminal Information Utilities.

You may also see

```
Sorry, I need to know a more specific terminal type
   than "unknown"
```

If this message appears, set and export the `TERM` as described in “Step 1: Set Up the Environment” on page 9-2.

Command Line Syntax for Editors

`cscope` invokes the **`vi`** editor by default. You may override the default setting by assigning your preferred editor to the `EDITOR` environment variable and exporting `EDITOR`, as described in the section “Step 1: Set Up the Environment” on page 9-2. Note, however, that **`cscope`** expects the editor it uses to have a command line syntax of the form

editor +linenum filename

as does **`vi`**. If the editor you want to use does not have this command line syntax, you must write an interface between **`cscope`** and the editor.

Suppose you want to use **ed**, for example. Because **ed** does not allow specification of a line number on the command line, you will not be able to use it to view or edit files with **cscope** unless you write a shell script (called **myedit** here) that contains the following line:

```
/usr/bin/ed $2
```

Now set the value of `EDITOR` to your shell script and export `EDITOR`:

```
EDITOR=myedit; export EDITOR
```

When **cscope** invokes the editor for the list item you have specified, for example, line 17 in **main.c**, it will invoke your shell script with the command line

```
myedit +17 main.c
```

myedit will discard the line number `($1)` and call **ed** correctly with the file name `($2)`. You will then have to execute the appropriate **ed** commands to display and edit the line because you will not be moved automatically to line 17 of the file.

10
Analyzing Your Code with lint

10
Analyzing Your Code with lint

Introduction to lint

`lint` checks for code constructs that may cause your C program not to compile, or to execute with unexpected results. **`lint`** issues every error and warning message produced by the C compiler. It also issues “**`lint`**-specific” warnings about potential bugs and portability problems.

In particular, **`lint`** compensates for separate and independent compilation of files in C by flagging inconsistencies in definition and use across files, including any libraries you have used. In a large project environment especially, where the same function may be used by different programmers in hundreds of separate modules of code, **`lint`** can help discover bugs that otherwise might be difficult to find. A function called with one less argument than expected, for example, looks at the stack for a value the call has never pushed, with results correct in one condition, incorrect in another, depending on whatever happens to be in memory at that stack location. By identifying dependencies like this one, and dependencies on machine architecture as well, **`lint`** can improve the reliability of code run on your machine or someone else's.

Options and Directives

`lint` is a static analyzer, which means that it cannot evaluate the run-time consequences of the dependencies it detects. Certain programs may contain hundreds of unreachable `break` statements, and **`lint`** will give a warning for each of them. The shear number of **`lint`** messages issued can be distracting. **`lint`**, however, provides command line options and directives to help suppress warnings you consider to be spurious.

NOTE

Directives are special comments embedded in the source text.

For the example we've cited here,

- You can invoke **`lint`** with the **`-b`** option to suppress all complaints about unreachable `break` statements;
- For a finer-grained control, you can precede any unreachable statement with the comment `/* NOTREACHED */` to suppress the diagnostic for that statement.

"Usage" on page 10-6 details options and directives and introduces the **`lint`** filter technique, which lets you tailor **`lint`**'s behavior even more finely to your project's needs. It also shows you how to use **`lint`** libraries to check your program for compatibility with the library functions you have called in it.

lint and the Compiler

Nearly five hundred diagnostic messages are issued by **`lint`**. However, this chapter only describes those **`lint`**-specific warnings that are not issued by the compiler. Additionally, this chapter lists diagnostics issued both by **`lint`** and the compiler that are capable of being suppressed only by **`lint`** options. For the text and examples of all messages issued exclusively by **`lint`** or subject exclusively to its options, refer to "lint-specific Messages" on page 10-12.

Message Formats

Most of **`lint`**'s messages are simple, one-line statements printed for each occurrence of the problem they diagnose. Errors detected in included files are reported multiple times by the compiler but only once by **`lint`**, no matter how many times the file is included in other source files. Compound messages are issued for inconsistencies across files and, in a few cases, for problems within them as well. A single message describes every occurrence of the problem in the file or files being checked. When use of a **`lint`** filter requires that a message be printed for each occurrence, compound diagnostics can be converted to the simple type by invoking **`lint`** with the **`-s`** option.

NOTE

See "Usage" on page 10-6 for more information.

What lint Does

`lint`-specific diagnostics are issued for three broad categories of conditions: inconsistent use, non-portable code, and suspicious constructs. In this section, we'll review examples of **`lint`**'s behavior in each of these areas, and suggest possible responses to the issues they raise.

Consistency Checks

Inconsistent use of variables, arguments, and functions is checked within files as well as across them. Generally speaking, the same checks are performed for prototype uses, declarations, and parameters as for old-style functions. (If your program does not use function prototypes, **`lint`** will check the number and types of parameters in each call to a function more strictly than the compiler.) **`lint`** also identifies mismatches of conversion specifications and arguments in `[fs]printf` and `[fs]scanf` control strings. Examples:

- Within files, **`lint`** flags non-`void` functions that "fall off the bottom" without returning a value to the invoking function. In the past, programmers often indicated that a function was not meant to return a value by omitting the return type: `fun() {}`. That convention means nothing to the compiler, which regards `fun` as having the return type `int`. Declare the function with the return type `void` to eliminate the problem.
- Across files, **`lint`** detects cases where a non-`void` function does not return a value, yet is used for its value in an expression, and the opposite problem, a function returning a value that is sometimes or always ignored in subsequent calls. When the value is always ignored, it may indicate an inefficiency in the function definition. When it is sometimes ignored, it's probably bad style (typically, not testing for error conditions). If you do not need to check the return values of string functions like `strcat`, `strcpy`, and `sprintf`, or output functions like `printf` and `putchar`, cast the offending call(s) to `void`.
- **`lint`** identifies variables or functions that are declared but not used or defined; used but not defined; or defined but not used. That means that when **`lint`** is applied to some, but not all files of a collection to be loaded together, it will complain about functions and variables declared in those files but defined or used elsewhere; used there but defined elsewhere; or defined there and used elsewhere. Invoke the **`-x`** option to suppress the former complaint, **`-u`** to suppress the latter two.

Portability Checks

Some non-portable code is flagged by **`lint`** in its default behavior, and a few more cases are diagnosed when **`lint`** is invoked with **`-p`** and/or **`-Xc`**. The latter tells **`lint`** to check for constructs that do not conform to the ANSI C standard. For the messages issued under **`-p`** and **`-Xc`**, check "Usage" on page 10-6. Examples:

- In some C language implementations, character variables that are not explicitly declared `signed` or `unsigned` are treated as signed quantities with a range typically from -128 to 127. In other implementations, they are treated as nonnegative quantities with a range typically from 0 to 255. So the test

```
char c;

c = getchar();
if (c == EOF) . . .
```

  where `EOF` has the value -1, will always fail on machines where character variables take on nonnegative values. One of **`lint`**'s **`-p`** checks will flag any comparison that implies a "plain" `char` may have a negative value. Note, however, that declaring `c` a `signed char` in the above example eliminates the diagnostic, not the problem. That's because `getchar` must return all possible characters and a distinct `EOF` value, so a `char` cannot store its value. This example, which is perhaps the most common one arising from implementation-defined sign-extension, shows how a thoughtful application of **lint**'s portability option can help you discover bugs not related to portability. In any case, declare `c` as an `int`.

- A similar issue arises with bit-fields. When constant values are assigned to bit-fields, the field may be too small to hold the value. On a machine that treats bit-fields of type `int` as unsigned quantities, the values allowed for `int x:3` range from 0 to 7, whereas on machines that treat them as signed quantities they range from -4 to 3. However unintuitive it may seem, a three-bit field declared type `int` cannot hold the value 4 on the latter machines. **lint** invoked with **-p** flags all bit-field types other than `unsigned int` or `signed int`. Note that these are the only portable bit-field types. The compilation system supports `int`, `char`, `short`, and `long` bit-field types that may be `unsigned`, `signed`, or "plain." It also supports the `enum` bit-field type.

- Bugs can arise when a larger-sized type is assigned to a smaller-sized type. If significant bits are truncated, accuracy is lost:

  ```
  short s;
  long l;
  s = l;
  ```

  lint flags all such assignments by default; the diagnostic can be suppressed by invoking the **-a** option. Bear in mind that you may be suppressing other diagnostics when you invoke **lint** with this or any other option. Check the list in "Usage" on page 10-6 for the options that suppress more than one diagnostic.

- A cast of a pointer to one object type to a pointer to an object type with stricter alignment requirements may not be portable. **lint** flags

  ```
  int *fun(y)
       char *y;
  {
       return(int *)y;
  }
  ```

  because, on most machines, an `int` cannot start on an arbitrary byte boundary, whereas a `char` can. If you suppress the diagnostic by invoking **lint** with **-h**, you may be disabling other messages. You can eliminate the problem by using the generic pointer `void *`.

- ANSI C leaves the order of evaluation of complicated expressions undefined. What this means is that when function calls, nested assignment statements, or the increment and decrement operators cause side effects — when a variable is changed as a by-product of the evaluation of an expression — the order in which the side effects take place is highly machine dependent. By default, **lint** flags any variable changed by a side effect and used elsewhere in the same expression:

  ```
  int a[10];
  main()
  {
      int i = 1;
      a[i++] = i;
  }
  ```

Note that in this example the value of `a[1]` may be 1 if one compiler is used, 2 if another. The bitwise logical operator & can also give rise to this diagnostic when it is mistakenly used in place of the logical operator &&:

```
    if ((c = getchar()) != EOF & c != '0')
```

Suspicious Constructs

`lint` flags a number of valid constructs that may not represent what the programmer intended. Examples:

- An `unsigned` variable always has a nonnegative value. So the test

  ```
      unsigned x;
      if (x < 0) . . .
  ```

  will always fail. Whereas the test

  ```
      unsigned x;
      if (x > 0) . . .
  ```

  is equivalent to

  ```
      if (x != 0) . . .
  ```

  which may not be the intended action. **`lint`** flags suspicious comparisons of `unsigned` variables with negative constants or 0. To compare an `unsigned` variable to the bit pattern of a negative number, cast it to `unsigned`:

  ```
      if (u == (unsigned) -1) . . .
  ```

  Or use the `U` suffix:

  ```
      if (u == -1U) . . .
  ```

- **`lint`** flags expressions without side effects that are used in a context where side effects are expected, where the expression may not represent what the programmer intended. It issues an additional warning whenever the equality operator is found where the assignment operator was expected, in other words, where a side effect was expected:

  ```
      int fun()
      {
           int a, b, x, y;
           (a = x) && (b == y);
      }
  ```

- **`lint`** cautions you to parenthesize expressions that mix both the logical and bitwise operators (specifically, `&, |, ^, <<, >>`), where misunderstanding of operator precedence may lead to incorrect results. Because the precedence of bitwise `&`, for example, falls below logical `==`, the expression

  ```
      if (x & a == 0) . . .
  ```

will be evaluated as

```
if (x & (a == 0)) . . .
```

which is most likely not what you intended. Invoking **lint** with **-h** disables the diagnostic.

Usage

You invoke **lint** with a command of the form

lint *file***.c** *file.c*

lint examines code in two passes. In the first, it checks for error conditions local to C source files, in the second for inconsistencies across them. This process is invisible to the user unless **lint** is invoked with **-c**:

```
lint -c file1.c file2.c
```

That command directs **lint** to execute the first pass only and collect information relevant to the second — about inconsistencies in definition and use across **file1.c** and **file2.c** — in intermediate files named **file1.ln** and **file2.ln**:

```
ls -1
file1.c
file1.ln
file2.c
file2.ln
```

In this way, the **-c** option to **lint** is analogous to the **-c** option to **cc**, which suppresses the link editing phase of compilation. Generally speaking, **lint**'s command line syntax closely follows **cc**'s.

When the **.ln** files are **lint**ed

```
lint file1.ln file2.ln
```

the second pass is executed. **lint** processes any number of **.c** or **.ln** files in their command line order. So

```
lint file1.ln file2.ln file3.c
```

directs **lint** to check **file3.c** for errors internal to it and all three files for consistency.

lint searches directories for included header files in the same order as **cc**

NOTE

For further information, see "Preprocessing Directives" in Concurrent *C Reference Manual*.

Use the **-I** option to **lint** as you would the **-I** option to **cc**. If you want **lint** to check an included header file that is stored in a directory other than your current directory or the standard place, specify the path of the directory with **-I** as follows:

lint -I*dir* **file1.c file2.c**

You can specify **-I** more than once on the **lint** command line. Directories are searched in the order they appear on the command line. Of course, you can specify multiple options to **lint** on the same command line. Options may be concatenated unless one of the options takes an argument:

lint -cp -I*dir* **-I***dir* **file1.c file2.c**

That command directs **lint** to

- Execute the first pass only;
- Perform additional portability checks;
- Search the specified directories for included header files.

lint Libraries

You can use **lint** libraries to check your program for compatibility with the library functions you have called in it: the declaration of the function return type, the number and types of arguments the function expects, and so on. The standard **lint** libraries correspond to libraries supplied by the C compilation system, and generally are stored in the standard place on your system, the directory **/usr/ccs/lib**. By convention, **lint** libraries have names of the form **llib-lx.ln**.

The **lint** standard C library, **llib-lc.ln**, is appended to the **lint** command line by default; checks for compatibility with it can be suppressed by invoking the **-n** option. Other **lint** libraries are accessed as arguments to **-l**.

lint -lx file1.c file2.c

directs **lint** to check the usage of functions and variables in **file1.c** and **file2.c** for compatibility with the **lint** library **llib-lx.ln**. The library file, which consists only of definitions, is processed exactly as are ordinary source files and ordinary.**ln** files, except that functions and variables used inconsistently in the library file, or defined in the library file but not used in the source files, elicit no complaints.

To create your own **lint** library, insert the directive `/* LINTLIBRARY */` at the head of a C source file, then invoke **lint** for that file with the **-o** option and the library name that will be given to **-l**:

lint -o*x files headed by* **/* LINTLIBRARY */**

causes only definitions in the source files headed by `/* LINTLIBRARY */` to be written to the file **llib-lx.ln**. (Note the analogy of **lint -o** to **cc -o**.) A library can be created from a file of function prototype declarations in the same way, except that both `/* LINTLIBRARY */` and `/* PROTOLIB`*n* `*/` must be inserted at the head of the declarations file. If *n* is 1, prototype declarations will be written to a library.**ln** file just as are old-style definitions. If *n* is 0, the default, the process is canceled. Invoking **lint** with **-y** is another way of creating a **lint** library:

```
lint -y -ox  file1.c file2.c
```

causes each source file named on the command line to be treated as if it began with `/* LINTLIBRARY */` and only its definitions to be written to **llib-lx.ln**.

By default, **lint** searches for **lint** libraries in the standard place. To direct **lint** to search for a **lint** library in a directory other than the standard place, specify the path of the directory with the **-L** option:

lint -L*dir* **-lx file1.c file2.c**

The specified directory is searched before the standard place.

lint Filters

A **lint** filter is a project-specific post-processor that typically uses an **awk** script or similar program to read the output of **lint** and discard messages that your project has decided do not identify real problems — string functions, for instance, returning values that are sometimes or always ignored. It enables you to generate customized diagnostic reports when **lint** options and directives do not provide sufficient control over output.

Two options to **lint** are particularly useful in developing a filter. Invoking **lint** with **-s** causes compound diagnostics to be converted into simple, one-line messages issued for each occurrence of the problem diagnosed. The easily parsed message format is suitable for analysis by an **awk** script.

Invoking **lint** with **-k** causes certain comments you have written in the source file to be printed in output, and can be useful both in documenting project decisions and specifying the post-processor's behavior. In the latter instance, if the comment identified an expected **lint** message, and the reported message was the same, the message might be filtered out. To use **-k**, insert on the line preceding the code you want to comment the `/* LINTED` [*msg*] `*/` directive, where *msg* refers to the comment to be printed when **lint** is invoked with **-k**. (Refer to the list of directives below for what **lint** does when **-k** is not invoked for a file containing `/* LINTED` [*msg*] `*/`.)

Options and Directives Listed

These options suppress specific messages:

-a Suppress:

- `assignment causes implicit narrowing conversion`
- `conversion to larger integral type may sign-extend incorrectly`

-b For unreachable `break` and empty statements, suppress:

- `statement not reached`

-h Suppress:

- `assignment operator “=” found where equality operator “==” was expected`
- `constant operand to op: “!”`
- `fallthrough on case statement`
- `pointer cast may result in improper alignment`
- `precedence confusion possible; parenthesize`
- `statement has no consequent: if`
- `statement has no consequent: else`

-m Suppress:

- `declared global, could be static`

-u Suppress:

- `name defined but never used`
- `name used but not defined`

-v Suppress:

- `argument unused in function`

-x Suppress:

- `name declared but never used or defined`

These options enable specific messages:

-p Enable:

- `conversion to larger integral type may sign-extend incorrectly`
- `may be indistinguishable due to truncation or case`
- `pointer casts may be troublesome`
- `nonportable bit-field type`
- `suspicious comparison of char with` *value*`: op “`*op*`”`

-Xc Enable:

- `bitwise operation on signed value nonportable`
- `function must return int: main()`
- `may be indistinguishable due to truncation or case`
- `only 0 or 2 parameters allowed: main()`
- `nonportable character constant`

Other options:

-c Create a **.ln** file consisting of information relevant to **lint**'s second pass for every **.c** file named on the command line. The second pass is not executed.

-F When referring to the **.c** files named on the command line, print their path names as supplied on the command line rather than only their base names.

-I*dir* Search the directory *dir* for included header files.

-k When used with the directive `/* LINTED [`*msg*`] */`, print `info:` *msg*.

-lx Access the **lint** library **llib-lx.ln**.

-L*dir* When used with **-l**, search for a **lint** library in the directory *dir*.

-n Suppress checks for compatibility with the default **lint** standard C library.

-ox Create the file **llib-lx.ln**, consisting of information relevant to **lint**'s second pass, from the **.c** files named on the command line. Generally used with **-y** or `/* LINTLIBRARY */` to create **lint** libraries.

-s Convert compound messages into simple ones.

-y Treat every **.c** file named on the command line as if it began with the directive `/* LINTLIBRARY */`.

-V Write the product name and release to standard error.

Directives:

`/* ARGSUSED`*n*` */`
Suppress:

- `argument unused in function`

for every argument but the first *n* in the function definition it precedes. Default is 0.

`/* CONSTCOND */`
Suppress:

- `constant in conditional context`
- `constant operand to op: "!"`

- `logical expression always false: op “&&”`
- `logical expression always true: op “||”`

for the constructs it precedes. Also `/* CONSTANTCONDITION */`.

`/* EMPTY */`
Suppress:

- `statement has no consequent: else`

  when inserted between the `else` and semicolon;

- `statement has no consequent: if`

when inserted between the controlling expression of the `if` and semicolon.

`/* FALLTHRU */`
Suppress:

- `fallthrough on case statement`

for the `case` statement it precedes. Also `/* FALLTHROUGH */`.

`/* LINTED [`*msg*`] */`
When **`-k`** is not invoked, suppress every warning pertaining to an intra-file problem except:

- `argument unused in function`
- `declaration unused in block`
- `set but not used in function`
- `static unused`
- `variable unused in function`

for the line of code it precedes. *msg* is ignored.

`/* LINTLIBRARY */`
When **`-o`** is invoked, write to a library **`.ln`** file only definitions in the **`.c`** file it heads.

`/* NOTREACHED */`
Suppress:

- `statement not reached`

for the unreached statements it precedes;

- `fallthrough on case statement`

for the case it precedes that cannot be reached from the preceding case;

- `function falls off bottom without returning value`

for the closing curly brace it precedes at the end of the function.

`/* PRINTFLIKE`*n* `*/`

Treat the *n*th argument of the function definition it precedes as a `[fs]printf` format string and issue:

- `malformed format string`

for invalid conversion specifications in that argument, and

- `function argument type inconsistent with format`
- `too few arguments for format`
- `too many arguments for format`

for mismatches between the remaining arguments and the conversion specifications. **`lint`** issues these warnings by default for errors in calls to `[fs]printf` functions provided by the standard C library.

`/* PROTOLIB`*n* `*/`

When *n* is 1 and `/* LINTLIBRARY */` is used, write to a library **`.ln`** file only function prototype declarations in the **`.c`** file it heads. Default is 0, canceling the process.

`/* SCANFLIKE`*n* `*/`

Same as `/* PRINTFLIKEn */` except that the *n*th argument of the function definition is treated as a `[fs]scanf` format string. By default, **`lint`** issues warnings for errors in calls to `[fs]scanf` functions provided by the standard C library.

`/* VARARGS`*n* `*/`

For the function whose definition it precedes, suppress:

- `function called with variable number of arguments`

for calls to the function with *n* or more arguments.

lint-specific Messages

This section lists alphabetically the warning messages issued exclusively by **`lint`** or subject exclusively to its options. The code examples illustrate conditions in which the messages are elicited. Note that some of the examples would elicit messages in addition to the one stated.

argument unused in function

Format: Compound

A function argument was not used. Preceding the function definition with `/* ARGSUSED`*n*` */` suppresses the message for all but the first *n* arguments; invoking **`lint`** with **`-v`** suppresses it for every argument.

```
1   int fun(int x, int y)
2   {
3       return x;
4   }
5   /* ARGSUSED1 */
6   int fun2(int x, int y)
7   {
8       return x;
9   }
============
argument unused in function
    (1) y in fun
```

array subscript cannot be > value: value

Format: Simple

The value of an array element's subscript exceeded the upper array bound.

```
1   int fun()
2   {
3       int a[10];
4       int *p = a;
5       while (p != &a[10])  /* using address is ok */
6           p++;
7       return a[5 + 6];
8   }
============
(7)  warning: array subscript cannot be > 9: 11
```

array subscript cannot be negative: value

Format: Simple

The constant expression that represents the subscript of a true array (as opposed to a pointer) had a negative value.

```
1   int f()
2   {
3       int a[10];
4       return a[5 * 2 / 10 - 2];
5   }
============
(4)  warning: array subscript cannot be negative: -1
```

assignment causes implicit narrowing conversion

Format: Compound

An object was assigned to one of a smaller type. Invoking **`lint`** with **`-a`** suppresses the message. So does an explicit cast to the smaller type.

```
1   void fun()
2   {
3      short s;
4      long l = 0;
5      s = l;
6   }
============
assignment causes implicit narrowing conversion
(5)
```

assignment of negative constant to unsigned type

Format: Simple

A negative constant was assigned to a variable of `unsigned` type. Use a cast or the `U` suffix.

```
1   void fun()
2   {
3      unsigned i;
4      i = -1;
5      i = -1U;
6      i = (unsigned) (-4 + 3);
7   }
============
(4) warning: assignment of negative constant to unsigned
type
```

assignment operator ?=? found where ?==? was expected

Format: Simple

An assignment operator was found where a conditional expression was expected. The message is not issued when an assignment is made to a variable using the value of a function call or in the case of string copying (see the example below). The warning is suppressed when **`lint`** is invoked with **`-h`**.

```
1   void fun()
2   {
3       char *p, *q;
4       int a = 0, b = 0, c = 0, d = 0, i;
5       i = (a = b) && (c == d);
6       i = (c == d) && (a = b);
7       if (a = b)
8          i = 1;
9       while (*p++ = *q++);
10      while (a = b);
11      while ((a = getchar()) == b);
12      if (a = foo()) return;
13  }
============
(5) warning: assignment operator "=" found where "=="
    was expected
(7) warning: assignment operator "=" found where "=="
    was expected
(10) warning: assignment operator "=" found where "=="
    was expected
```

bitwise operation on signed value nonportable

Format: Compound

The operand of a bitwise operator was a variable of signed integral type, as defined by ANSI C. Because these operators return values that depend on the internal representations of integers, their behavior is implementation-defined for operands of that type. The message is issued only when **`lint`** is invoked with **`-Xc`**.

```
1   fun()
2   {
3       int i;
4       signed int j;
5       unsigned int k;
6       i = i & 055;
7       j = j | 022;
8       k = k >> 4;
9   }
============
warning: bitwise operation on signed value nonportable
    (6)       (7)
```

constant in conditional context

Format: Simple

The controlling expression of an `if`, `while`, or `for` statement was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message.

```
1   void fun()
2   {
3      if (! 1) return;
4      while (1) foo();
5      for (;1;);
6      for (;;);
7      /* CONSTCOND */
8      while (1);
9   }
============
(3) warning: constant in conditional context
(4) warning: constant in conditional context
(5) warning: constant in conditional context
```

constant operand to op: ?!?

Format: Simple

The operand of the NOT operator was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message for that statement; invoking **`lint`** with **`-h`** suppresses it for every statement.

```
1   void fun()
2   {
3      if  (! 0) return;
4      /* CONSTCOND */
5      if  (! 0) return;
6   }
============
(3) warning: constant operand to op: "!"
```

constant truncated by assignment

Format: Simple

An integral constant expression was assigned or returned to an object of an integral type that cannot hold the value without truncation.

```
1   unsigned char f()
2   {
3       unsigned char i;
4       i = 255;
5       i = 256;
6       return 256;
7   }
============
(5) warning: constant truncated by assignment
(6) warning: constant truncated by assignment
```

conversion of pointer loses bits

Format: Simple

A pointer was assigned to an object of an integral type that is smaller than the pointer.

```
1   void fun()
2   {
3       char c;
4       int *i;
5       c = i;
6   }
============
(5) warning: conversion of pointer loses bits
```

conversion to larger integral type may sign-extend incorrectly

Format: Compound

A variable of type “plain” `char` was assigned to a variable of a larger integral type. Whether a “plain” `char` is treated as signed or unsigned is implementation-defined. The message is issued only when **`lint`** is invoked with **`-p`**, and is suppressed when it is invoked with **`-a`**.

```
1   void fun()
2   {
3       char c = 0;
4       short s = 0;
5       long l;
6       l = c;
7       l = s;
8   }
============
conversion to larger integral type may sign-extend
incorrectly
    (6)
```

declaration unused in block

Format: Compound

An external variable or function was declared but not used in an inner block.

```
1   int fun()
2   {
3       int foo();
4       int bar();
5       return foo();
6   }
============
declaration unused in block
    (4) bar
```

declared global, could be static

Format: Compound

An external variable or function was declared global, instead of `static`, but was referenced only in the file in which it was defined. The message is suppressed when **`lint`** is invoked with **`-m`**.

file **`f1.c`**

```
1   int i;
2   int foo() {return i;}
3   int fun() {return i;}
4   static int stfun() {return fun();}
```

file **`f2.c`**

```
1   main()
2   {
3     int a;
4     a = foo();
5   }
============
declared global, could be static
    fun          f1.c(3)
    i            f1.c(1)
```

equality operator ?==? found where ?=? was expected

Format: Simple

An equality operator was found where a side effect was expected.

```
1   void fun(a, b)
2   int a, b;
3   {
4       a == b;
5       for (a == b; a < 10; a++);
6   }
============
(4) warning: equality operator "==" found where "="
    was expected
(5) warning: equality operator "==" found where "="
    was expected
```

evaluation order undefined: name

Format: Simple

A variable was changed by a side effect and used elsewhere in the same expression.

```
1   int a[10];
2   main()
3   {
4       int i = 1;
5       a[i++] = i;
6   }
============
(5) warning: evaluation order undefined: i
```

fallthrough on case statement

Format: Simple

Execution fell through one case to another without a `break` or `return`. Preceding a `case` statement with `/* FALLTHRU */`, or `/* NOTREACHED */` when the case cannot be reached from the preceding case (see below), suppresses the message for that statement; invoking **`lint`** with **`-h`** suppresses it for every statement.

```
1   void fun(i)
2   {
3      switch (i) {
4      case 10:
5          i = 0;
6      case 12:
7          return;
8      case 14:
9          break;
10     case 15:
11     case 16:
12         break;
13     case 18:
14         i = 0;
15          /* FALLTHRU */
16      case 20:
17          error("bad number");
18          /* NOTREACHED */
19      case 22:
20          return;
21      }
22  }
============
(6) warning: fallthrough on case statement
```

function argument (number) declared inconsistently

Format: Compound

The parameter types in a function prototype declaration or definition differed from their types in another declaration or definition. The message described after this one is issued for uses (not declarations or definitions) of a prototype with the wrong parameter types.

file **i3a.c**

```
1   int fun1(int);
2   int fun2(int);
3   int fun3(int);
```

file **i3b.c**

```
1   int fun1(int *i);
2   int fun2(int *i) {}
3   void foo()
4   {
5       int *i;
6       fun3(i);
7   }
============
function argument ( number ) declared inconsistently
    fun2 (arg 1)           i3b.c(2) int * :: i3a.c(2) int
    fun1 (arg 1)           i3a.c(1) int :: i3b.c(1) int *
function argument ( number ) used inconsistently
    fun3 (arg 1)           i3a.c(3) int :: i3b.c(6) int *
```

function argument (number) used inconsistently

Format: Compound

The argument types in a function call did not match the types of the formal parameters in the function definition. (And see the discussion of the preceding message.)

```
file f1.c
1   int fun(int x, int y)
2   {
3       return x + y;
4   }
file f2.c
1   int main()
2   {
3       int *x;
4       extern int fun();
5       return fun(1, x);
6   }
============
function argument ( number ) used inconsistently
    fun( arg 2 )        f1.c(2) int :: f2.c(5) int *
```

function argument type inconsistent with format

Format: Compound

An argument was inconsistent with the corresponding conversion specification in the control string of a **`[fs]printf`** or **`[fs]scanf`** function call. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in “Usage” on page 10-6.)

```
1   #include <stdio.h>
2   main()
3   {
4       int i;
5       printf("%s", i);
6   }
============
function argument type inconsistent with format
    printf(arg 2) int :: (format) char *  test.c(5)
```

function called with variable number of arguments

Format: Compound

A function was called with the wrong number of arguments. Preceding a function definition with `/* VARARGS`*n*` */` suppresses the message for calls with *n* or more arguments; defining and declaring a function with the ANSI C notation “`. . .`” suppresses it for every argument.

NOTE

See “function declared with variable number of arguments” on page 10-22 for more information.

file **`f1.c`**
```
1   int fun(int x, int y, int z)
2   {
3       return x + y + z;
4   }
5   int fun2(int x, . . .)
6   {
7       return x;
8   }
10  /* VARARGS1 */
11  int fun3(int x, int y, int z)
12  {
13      return x;
14  }
```
file **`f2.c`**
```
1   int main()
2   {
3       extern int fun(), fun3(), fun2(int x, . . .);
4       return fun(1, 2);
5       return fun2(1, 2, 3, 4);
6       return fun3(1, 2, 3, 4, 5);
7   }
============
function called with variable number of arguments
    fun         f1.c(2) :: f2.c(4)
```

function declared with variable number of arguments

Format: Compound

The number of parameters in a function prototype declaration or definition differed from their number in another declaration or definition. Declaring and defining the prototype with the ANSI C notation “`. . .`” suppresses the warning if all declarations have the same number of arguments. The message immediately preceding this one is issued for uses (not declarations or definitions) of a prototype with the wrong number of arguments.

file **`i3a.c`**
```
1   int fun1(int);
2   int fun2(int);
3   int fun3(int);
```

```
file i3b.c
1   int fun1(int, int);
2   int fun2(int a, int b) {}
3   void foo()
4   {
5       int i, j, k;
6       i = fun3(j, k);
7   }
============
function declared with variable number of arguments
    fun2           i3a.c(2) :: i3b.c(2)
    fun1           i3a.c(1) :: i3b.c(1)
function called with variable number of arguments
    fun3           i3a.c(3) :: i3b.c(6)
```

function falls off bottom without returning value

Format: Compound

A non-`void` function did not return a value to the invoking function. If the closing curly brace is truly not reached, preceding it with `/* NOTREACHED */` suppresses the message.

```
1   fun()
2   {}
3   void fun2()
4   {}
5   foo()
6   {
7      exit(1);
8   /* NOTREACHED */
9   }
============
function falls off bottom without returning value (2) fun
```

function must return int: main()

Format: Simple

The program's `main` function does not return `int`, in violation of ANSI C restrictions. The message is issued only when **`lint`** is invoked with **`-Xc`**.

```
1   void main()
2   {}
============
(2) warning: function must return int: main()
```

function returns pointer to [automatic/parameter]

Format: Simple

A function returned a pointer to an automatic variable or a parameter. Since an object with automatic storage duration is no longer guaranteed to be reserved after the end of the block, the value of the pointer to that object will be indeterminate after the end of the block.

```
1   int *fun(int x)
2   {
3       int a[10];
4       int b;
5       if (x == 1)
6          return a;
7       else if (x == 2)
8          return &b;
9       else return &x;
10  }
============
(6) warning: function returns pointer to automatic
(8) warning: function returns pointer to automatic
(9) warning: function returns pointer to parameter
```

function returns value that is always ignored

Format: Compound

A function contained a `return` statement and every call to the function ignored its return value.

file **`f1.c`**

```
int fun()
{
     return 1;
 }
```

file **`f2.c`**

```
extern int fun();
int main()
{
    fun();
    return 1;
}
============
function returns value that is always ignored fun
```

function returns value that is sometimes ignored

Format: Compound

A function contained a `return` statement and some, but not all, calls to the function ignored its return value.

file **f1.c**

```
int fun()
{
    return 1;
}
```

file **f2.c**

```
extern int fun();
int main()
{
    if(1) {
        return fun();
    }
else {
        fun();
        return 1;
    }
}
============
function returns value that is sometimes ignored
    fun
```

function value is used, but none returned

Format: Compound

A non-`void` function did not contain a `return` statement, yet was used for its value in an expression.

file **f1.c**

```
extern int fun();
main()
{
    return fun();
}
```

file **f2.c**

```
int fun()
{}
============
function value is used, but none returned
    fun
```

logical expression always false: op ?&&?

Format: Simple

A logical AND expression checked for equality of the same variable to two different constants, or had the constant 0 as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```
1   void fun(a)
2   int a;
3   {
4       a = (a == 1) && (a == 2);
5       a = (a == 1) && (a == 1);
6       a = (1 == a) && (a == 2);
7       a = (a == 1) && 0;
8       /* CONSTCOND */
9       a = (0 && (a == 1));
10  }
============
(4) warning: logical expression always false: op "&&"
(6) warning: logical expression always false: op "&&"
(7) warning: logical expression always false: op "&&"
```

logical expression always true: op ?||?

Format: Simple

A logical OR expression checked for inequality of the same variable to two different constants, or had a nonzero integral constant as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```
1   void fun(a)
2   int a;
3   {
4       a = (a != 1) || (a != 2);
5       a = (a != 1) || (a != 1);
6       a = (1 != a) || (a != 2);
7       a = (a == 10) || 1;
8       /* CONSTCOND */
9       a = (1 || (a == 10));
10  }
============
(4) warning: logical expression always true: op "||"
(6) warning: logical expression always true: op "||"
(7) warning: logical expression always true: op "||"
```

malformed format string

Format: Compound

A `[fs]printf` or `[fs]scanf` control string was formed incorrectly. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in “Usage” on page 10-6.)

```
1   #include <stdio.h>
2   main()
3   {
4       printf("%y");
5   }
============
malformed format string
    printf      test.c(4)
```

may be indistinguishable due to truncation or case

Format: Compound

External names in a program may be indistinguishable when it is ported to another machine due to implementation-defined restrictions as to length or case. The message is issued only when **`lint`** is invoked with **`-Xc`** or **`-p`**. Under **`-Xc`**, external names are truncated to the first 6 characters with one case, in accordance with the ANSI C lower bound; under **`-p`**, to the first 8 characters with one case.

```
file f1.c
1   int foobar1;
2   int FooBar12;
file f2.c
1   int foobar2;
2   int FOOBAR12;
============
under -p
may be indistinguishable due to truncation or case
    FooBar12   f1.c(2)  ::  FOOBAR12   f2.c(2)
under -Xc
may be indistinguishable due to truncation or case
    foobar1    f1.c(1)  ::  FooBar12   f1.c(2)
    foobar1    f1.c(1)  ::  foobar2    f2.c(1)
    foobar1    f1.c(1)  ::  FOOBAR12    f2.c(2)
```

name declared but never used or defined

Format: Compound

A non-`static` external variable or function was declared but not used or defined in any file. The message is suppressed when **`lint`** is invoked with **`-x`**.

file **`f.c`**

```
1   extern int fun();
2   static int foo();
============
name declared but never used or defined
    fun            f.c(1)
```

name defined but never used

Format: Compound

A variable or function was defined but not used in any file. The message is suppressed when **`lint`** is invoked with **`-u`**.

file **`f.c`**

```
1   int i, j, k = 1;
2   main()
3   {
4        j = k;
5   }
============
name defined but never used
    i           f.c(1)
```

name multiply defined

Format: Compound

A variable was defined in more than one source file.

file **`f1.c`**

```
1   char i = 'a';
```

file **`f2.c`**

```
1   long i = 1;
============
name multiply defined
    i           f1.c(1) :: f2.c(1)
```

name used but not defined

Format: Compound

A non-`static` external variable or function was declared but not defined in any file. The message is suppressed when **`lint`** is invoked with **`-u`**.

file **`f.c`**

```
1   extern int fun();
2   int main()
3   {
4       return fun();
5   }
============
name used but not defined
    fun         f.c(4)
```

nonportable bit-field type

Format: Simple

A bit-field type other than `signed int` or `unsigned int` was used. The message is issued only when **`lint`** is invoked with **`-p`**. Note that these are the only portable bit-field types. The compilation system supports `int`, `char`, `short`, and `long` bit-field types that may be `unsigned`, `signed`, or “plain.” It also supports the `enum` bit-field type.

```
1   struct u {
2       unsigned v:1;
3       int      w:1;
4       char     x:8;
5       long     y:8;
6       short    z:8;
7   };
============
(3) warning: nonportable bit-field type
(4) warning: nonportable bit-field type
(5) warning: nonportable bit-field type
(6) warning: nonportable bit-field type
```

nonportable character constant

Format: Simple

A multi-character character constant in the program may not be portable. The message is issued only when **`lint`** is invoked with **`-Xc`**.

```
1   int c = 'abc';
============
(1) warning: nonportable character constant
```

only 0 or 2 parameters allowed: main()

Format: Simple

The function `main` in your program was defined with only one parameter or more than two parameters, in violation of the ANSI C requirement. The message is issued only when **`lint`** is invoked with **`-Xc`**.

```
1   main(int argc, char **argv, char **envp)
2   {}
============
(2) warning: only 0 or 2 parameters allowed: main()
```

pointer cast may result in improper alignment

Format: Compound

A pointer to one object type was cast to a pointer to an object type with stricter alignment requirements. Doing so may result in a value that is invalid for the second pointer type. The warning is suppressed when **`lint`** is invoked with **`-h`**.

```
1   void fun()
2   {
3       short *s;
4       int *i;
5       i = (int *) s;
6   }
============
pointer cast may result in improper alignment
    (5)
```

pointer casts may be troublesome

Format: Compound

A pointer to one object type was cast to a pointer to a different object type. The message is issued only when **`lint`** is invoked with **`-p`**, and is not issued for the generic pointer `void *`.

```
1   void fun()
2   {
3       int *i;
4       char *c;
5       void *v;
6       i = (int *) c;
7       i = (int *) v;
8   }
============
warning: pointer casts may be troublesome
    (6)
```

precedence confusion possible; parenthesize

Format: Simple

An expression that mixes a logical and a bitwise operator was not parenthesized. The message is suppressed when **lint** is invoked with **-h**.

```
1   void fun()
2   {
3       int x = 0, m = 0, MASK = 0, i;
4       i = (x + m == 0);
5       i = (x & MASK == 0);   /* eval'd
                                 (x & (MASK == 0)) */
6       i = (MASK == 1 & x);  /* eval'd
                                 ((MASK == 1) & x) */
7   }
============
(5) warning: precedence confusion possible; parenthesize
(6) warning: precedence confusion possible; parenthesize
```

precision lost in bit-field assignment

Format: Simple

A constant was assigned to a bit-field too small to hold the value without truncation. Note that in the following example the bit-field `z` may have values that range from 0 to 7 or -4 to 3, depending on the machine.

```
1   void fun()
2   {
3       struct {
4           signed x:3;     /* max value allowed is 3 */
5           unsigned y:3;   /* max value allowed is 7 */
6           int z:3;        /* max value allowed is 7 */
7       } s;
8       s.x = 3;
9       s.x = 4;
10      s.y = 7;
11      s.y = 8;
12      s.z = 7;
13      s.z = 8;
14  }
============
(9) warning: precision lost in bit-field assignment: 4
(11) warning: precision lost in bit-field assignment: 0x8
(13) warning: precision lost in bit-field assignment: 8
```

set but not used in function

Format: Compound

An automatic variable or a function parameter was declared and set but not used in a function.

```
1   void fun(y)
2   int y;
3   {
4       int x;
5       x = 1;
6       y = 1;
7   }
============
set but not used in function
    (4) x in fun
    (1) y in fun
```

statement has no consequent: else

Format: Simple

An `if` statement had a null `else` part. Inserting `/* EMPTY */` between the `else` and semicolon suppresses the message for that statement; invoking **`lint`** with **`-h`** suppresses it for every statement.

```
1   void f(a)
2   int a;
3   {
4       if (a)
5          return;
6       else;
7   }
============
(6) warning: statement has no consequent: else
```

statement has no consequent: if

Format: Simple

An `if` statement had a null `if` part. Inserting `/* EMPTY */` between the controlling expression of the `if` and semicolon suppresses the message for that statement; invoking **`lint`** with **`-h`** suppresses it for every statement.

```
1   void f(a)
2   int a;
3   {
4       if (a);
5       if (a == 10)
6          /* EMPTY */;
7       else return;
8   }
============
(4) warning: statement has no consequent: if
```

statement has null effect

Format: Compound

An expression did not generate a side effect where a side effect was expected. Note that the message is issued for every subsequent sequence point that is reached at which a side effect is not generated.

```
1   void fun()
2   {
3       int a, b, c, x;
4       a;
5       a == 5;
6       ;
7       while (x++ != 10);
8       (a == b) && (c = a);
9       (a = b) && (c == a);
10      (a, b);
11  }
============
statement has null effect
    (4)           (5)           (9)           (10)
```

statement not reached

Format: Compound

A function contained a statement that cannot be reached. Preceding an unreached statement with `/* NOTREACHED */` suppresses the message for that statement; invoking **`lint`** with **`-b`** suppresses it for every unreached `break` and empty statement. Note that this message is also issued by the compiler but cannot be suppressed.

```
1   void fun(a)
2   {
3       switch (a) {
4           case 1:
5              return;
6              break;
7           case 2:
8              return;
9              /* NOTREACHED */
10             break;
11      }
12  }
============
statement not reached
    (6)
```

static unused

Format: Compound

A variable or function was defined or declared `static` in a file but not used in that file. Doing so is probably a programming error because the object cannot be used outside the file.

```
1   static int x;
2   static int main() {}
3   static int foo();
4   static int y = 1;
============
static unused
    (4) y        (3) foo        (2) main        (1) x
```

suspicious comparison of char with value: op ?op?

Format: Simple

A comparison was performed on a variable of type “plain” `char` that implied it may have a negative value (< 0, <= 0, >= 0, > 0). Whether a “plain” `char` is treated as signed or non-negative is implementation-defined. The message is issued only when **`lint`** is invoked with **`-p`**.

```
1   void fun(c, d)
2   char c;
3   signed char d;
4   {
5       int i;
6       i = (c == -5);
7       i = (c < 0);
8       i = (d < 0);
9   }
```

```
============
(6) warning: suspicious comparison of char with negative
    constant: op "=="
(7) warning: suspicious comparison of char with 0: op "<"
```

suspicious comparison of unsigned with value: op ?op?

Format: Simple

A comparison was performed on a variable of `unsigned` type that implied it may have a negative value (< 0, <= 0, >= 0, > 0).

```
1   void fun(x)
2   unsigned x;
3   {
4       int i;
5       i = (x > -2);
6       i = (x < 0);
7       i = (x <= 0);
8       i = (x >= 0);
9       i = (x > 0);
10      i = (-2 < x);
11      i = (x == -1);
12      i = (x == -1U);
13  }
============
(5) warning: suspicious comparison of unsigned with
    negative constant: op ">"
(6) warning: suspicious comparison of unsigned with 0:
    op "<"
(7) warning: suspicious comparison of unsigned with 0:
    op "<="
(8) warning: suspicious comparison of unsigned with 0:
    op ">="
(9) warning: suspicious comparison of unsigned with 0:
    op ">"
(10) warning: suspicious comparison of unsigned with
    negative constant: op "<"
(11) warning: suspicious comparison of unsigned with
    negative constant: op "=="
```

too few arguments for format

Format: Compound

A control string of a `[fs]printf` or `[fs]scanf` function call had more conversion specifications than there were arguments remaining in the call. (See also `/* PRINTF-LIKE`*n* `*/` and `/* SCANFLIKE`*n* `*/` in the list of directives in “Usage” on page 10-6.)

```
1   #include <stdio.h>
2   main()
3   {
4       int i;
5       printf("%d%d", i);
6   }
============
too few arguments for format
    printf      test.c(5)
```

too many arguments for format

Format: Compound

A control string of a `[fs]printf` or `[fs]scanf` function call had fewer conversion specifications than there were arguments remaining in the call. (See also `/* PRINTF-LIKE`*n* `*/` and `/* SCANFLIKE`*n* `*/` in the list of directives in “Usage” on page 10-6.)

```
1   #include <stdio.h>
2   main()
3   {
4       int i, j;
5       printf("%d", i, j);
6   }
============
too many arguments for format
    printf      test.c(5)
```

value type declared inconsistently

Format: Compound

The return type in a function declaration or definition did not match the return type in another declaration or definition of the function. The message is also issued for inconsistent declarations of variable types.

file **`f1.c`**

```
1   void fun() {}
2   void foo();
3   extern int a;
```

file **`f2.c`**

```
1   extern int fun();
2   extern int foo();
3   extern char a;
============
value type declared inconsistently
    fun         f1.c(1) void() :: f2.c(1) int()
    foo         f1.c(2) void() :: f2.c(2) int()
    a           f1.c(3) int :: f2.c(3) char
```

value type used inconsistently

Format: Compound

The return type in a function call did not match the return type in the function definition.

file **`f1.c`**

```
1   int *fun(p)
2   int *p;
3   {
4       return p;
5   }
```

file **`f2.c`**

```
1   main()
2   {
3       int i, *p;
4       i = fun(p);
5   }
============
value type used inconsistently
    fun        f1.c(3) int *() :: f2.c(4) int()
```

variable may be used before set: name

Format: Simple

The first reference to an automatic, non-array variable occurred at a line number earlier than the first assignment to the variable. Note that taking the address of a variable implies both a set and a use, and that the first assignment to any member of a `struct` or `union` implies an assignment to the entire `struct` or `union`.

```
1   void fun()
2   {
3       int i, j, k;
4       static int x;
5       k = j;
6       i = i + 1;
7       x = x + 1;
8   }
============
(5) warning: variable may be used before set: j
(6) warning: variable may be used before set: i
```

variable unused in function

Format: Compound

A variable was declared but never used in a function.

```
1   void fun()
2   {
3       int x, y;
4       static z;
5   }
============
variable unused in function
    (4) z in fun
    (3) y in fun
    (3) x in fun
```

11
Performance Analysis

11
Performance Analysis

Introduction

An analysis of the run-time performance and characteristics of a program can identify sections of code which have a significant effect on the speed and behavior of the program. PowerUX provides a tool which can be used to obtain an execution profile of <u>any</u> program.

Two traditional UNIX tools provide profile data for a program which has been compiled to produce this data during execution. The output from **`prof`** identifies which routines in the program have been executed, how often they were invoked, and what percentage of the program's execution time was spent in each routine. The **`gprof`** tool additionally provides a call graph of the ancestors and descendants of the routines. These tools are not available on supported hardware platforms.

Better information can be obtained through a Concurrent-developed tool, called **`analyze`**. Where **`prof`** and **`gprof`** require a special compilation of a program for producing profile data, **`analyze`** operates on already-compiled code. It may be necessary to invoke **`analyze`** through a link edit step, but recompilation of the program is not necessary. **`analyze`** first reads an executable file. It then interprets the instructions in this file to find the routines and basic blocks (a block is a sequence of instructions having one entry and one exit point) within each routine. Next, **`analyze`** performs a local timing analysis for each basic block to determine statistics like; the time spent in the block, or places where execution is delayed due to pipe constraints, etc. A companion tool, **`report`**, produces information that is useful in evaluating the program's performance.

`analyze` can also be used to transform, or even eliminate instructions in the program, to produce <u>faster</u> running code. Thus, it is able to further optimize code that has previously been compiled.

analyze

Information

The lowest level of detailed output is generated with the **`-d`** option, which generates a disassembly listing, and the **`-v`** option, which annotates that listing with detailed information on the resources being used.

Because there is so much information, it is compressed into a fairly cryptic form:

`t=#`	This indicates the relative clock time. Everything on the same line happens at the same time.
`u#r`	An entry that starts with the letter `u` indicates a resource is now being used. The number following the `u` is the sequence number of the instruction within the basic block that is using the resource, finally the resource name appears immediately following the number (resource names are things like registers or pipeline stages).
`f#r`	An entry that starts with `f` indicates the instruction at the given sequence has now freed the resource.
`b#r[#]`	The `b` entry indicates an instruction that has been blocked because it needs a resource. The number at the end enclosed in brackets is the sequence number of the instruction which currently has the resource and is the cause of the block.
`s#r`	On the PowerPC platforms, individual pipeline stages are not shown as allocated and freed. Instead, it is simply announced that a particular instruction has entered a particular stage with the `s` entry.

Use the **`-Zstage_status`** option to cause **`analyze`**'s output to include the status of all the pipeline stages each cycle. While this output is much easier to read, it is extremely verbose. Note that instructions are disassembled at the cycle they enter dispatch. Negative numbers in the **`stage_status`** output are placeholders for pipeline bubbles caused by alignment constraints. Screen 11-1 illustrates this situation.

```
10 (10001028) 3d000000   lis r8,0           t=6
11 (1000102c) 39200000   li r9,0                    t=6
   t=6
        Fetch:   16   17   18   19
        Decode:   12   13   14   15
        Dispatch:    8    9   10   11
               SCIU1   SCIU2        MCIU        FPU    LSU    BPU
        Q2     ----     ----        ----       ----   ----   ----
        Q1     ----     ----    spr ---- m/d   ----   ----   ----
        X1        6        7   ----      ----  ----   ----   ----
        X2                     ----      ----  ----   ----
        X3                               ----  ----   ----
        Finished:    4    5
        Complete:    4    5
        Writeback   GPR: 4 5   FPR:   CRF:
```

Screen 11-1. Sample Output from analyze

When **`analyze`** prints an instruction out, it puts it on a line by itself with the clock time it started execution on the end. The fields on the line represent the source line number (blank if no debug information is available in the file), the sequence number within the block, the absolute address of the instruction in the file, the four-byte hex for the instruction itself, then the symbolic disassembly of the instruction.

Currently, max time is defined as the total number of cycles required for all instructions in the block to make it through all pipe stages. It, therefore, represents a worst-case upper bound.

Note that all times are local; a block containing a subroutine call will only have the time for the call instruction. No information is computed about the time actually spent in the subroutine, and no information is known about the state of the pipelines when the subroutine returns. The max time for a block ending in a subroutine call does not count any cycles remaining in the pipe at the time the call is made because most of these cycles never cause any delay (the subroutine is usually still in the prologue when the pipe drains).

The optimization features of **`analyze`** can be invoked at link edit time by using the Concurrent link editor's **`-O`** option. Refer to **`ld(1)`** and Chapter 20 ("Program Optimization"), for more information.

Statistics

The **`analyze`** tool computes several statistics, some of which are more meaningful than others, but all are designed to help someone analyze the quality of generated code.

BURT

BURT stands for *Bogus Uniform Routine Time*, and (as its name indicates) is a fairly bogus statistic which may have some value as a guide. It is computed by multiplying the max time for each basic block by a weighting factor that increases rapidly as the loop nesting level goes up. The accumulated time for all the blocks is the `BURT` number.

ERNIE

ERNIE is *External Routine Necessary Interface Executions*, and is a statistic designed to help you decide if `BURT` numbers are different because subroutines have been inlined (or vice-versa), or if they are different simply because of different code quality. `ERNIE` is computed by simply adding up all the nesting level factors for any block that contains a subroutine call.

The above statistics all depend on accurately computing loop nesting levels. If the flow graph is irreducible, then it is difficult to decide just what a loop is, so a warning is generated for routines with irreducible flow graphs. Often when code finally gets generated, a single basic block will be the header of several back edges. Each back edge is counted as a separate loop, so the nesting level for the header may get very high.

Profiling

The **`-P`** option patches the input program, generating a new program which will accumulate cycle count statistics at the basic block level and dump them to an output file on exit. The statistics are always dumped to a file with the same name as the executable given as the argument to **`-P`**, with the **`.prof`** suffix added. For example, if you specified **`-P`***fred* then when you run the generated program the file **`fred.prof`** will be generated with the profiling statistics.

The **`-C`** option adds statistics about cache misses due to instruction fetches and data accesses to the profile data. With the **`-C`** option, the patched program simulates the activity of the primary instruction and data caches, as well as that of the secondary cache. This option can be useful for diagnosing performance problems arising from lack of memory access locality (proximity). It should be used with care because it can significantly increase run-time overhead and the size of the executable program.

Currently, the statistics are only as accurate as the timing information shown in the disassembly listing. Both min and max times are accumulated, so the report can print only upper and lower bounds on the cycle count. A future version may attempt to add code that will correct the cycle count with additional information gathered about pipe conflicts that will occur depending on the arc followed to reach each basic block.

It is often difficult to profile some programs, especially those generated by non-Concurrent compilers. The following guidelines are given as an aid to people attempting to profile foreign code:

The **`analyze`** tool relies on the symbol table to find subroutine entry points. A stripped program cannot be profiled. Even if a symbol table exists, **`analyze`** can identify subroutine entry points only if they have associated tdesc information, if they have symbolic debug information identifying them as subroutine entry points, or if they are explicitly named using the **`-a`** option.

`analyze` records its profile statistics by writing them into the **`.bss`** section. The header of the object file is modified to reserve space in **`.bss`**, but the run-time environment also needs to be informed that the space is being used. **`analyze`** does this by first attempting to patch the initial value of the global variable (`curbrk`) used by the library routines to record the break address. If this variable is not found in the symbol table it then attempts to patch a call to `brk()` into the main entry point. If it cannot find the `brk()` entry point in the symbol table, then it cannot successfully patch the program. It may be necessary to re-link the program, forcing the `brk()` routine to be included by linking in an additional object file that references it, or use the **`-Zbreak=`***name* option to specify a different name for the break variable.

Finally, **`analyze`** writes the statistics out by patching in a call to the write routine when the `__exit` routine is called (that is two underscores). If the low level exit routine is not called `__exit` or if the program exits in a different way (possibly by calling `exec()`), then you will need to use the **`-X`** option to name the routines that should dump statistics.

After dumping the statistics at an exit point, all the basic block counts are set to zero. This feature allows you to divide your program into separate sections which will be profiled independently, each generating a separate data set in the **`.prof`** file. All you need to do is call a dummy routine once between each section of the program, then use the **`-X`** option to declare these dummy routines as exit points.

If any basic block begins with a trap instruction of some kind, **`analyze`** will generate a warning. Normally it relies on the flow of control resuming right after the patched instruction, but it is uncertain where control will resume after the kernel gets control. Unless you know what the routine does, it might be wise to exclude it from the list of routines to be profiled.

Usage

`analyze` is invoked as follows:

analyze [-A] [-C] [-D *flag***] [-H] [-N] [-O** *file***] [-P** *file***]**
[-S *section***] [-X** *routine***] [-W** *routine***[=***weight***]] [-a** *routine***]**
[-d *file***] [-g** *file***] [-i] [-n] [-r** *file***] [-s** *routine***]**
[-v] [-x] [-Z *keyword***]** *file*

The *file* argument specifies the name of the executable file over which **analyze** will be run. All other arguments are optional and are as follows:

-A Include all the routines in the analysis. This is the default mode of operation.

-C Gather cache activity statistics during profiling. This option works with **-P** and has no effect without it. It also writes its statistics to *file***.prof** as specified by the **-P** option. Cache statistics include instruction accesses gathered at each basic block, and data accesses gathered at each load or store instruction. The **report** program can be used to generate various reports that include this information.

-D *flag* Turn on the specified debug flag. You will not be interested in using this unless you know a lot about the inner details of **analyze**.

-H Print a summary of the command usage.

-N Set the list of routines to be analyzed to the empty set. This overrides the default setting (which corresponds to **-A** above).

-O *file* Generate a new program file in *file* which has been optimized by replacing many of the two-instruction sequences (which are required to reference global memory locations) with single instructions which use the reserved linker registers (`r28` through `r31`) as base registers. This allows faster access to the four most commonly referenced 64K data blocks. Certain library routines that are known to access the linker registers (e.g., `setjmp` and `longjmp`) are automatically excluded from the optimization process. The **-X** option may be used to specifically exclude others. (Normally any reference to a linker register will cause an error).

-P *file* Generate a new program file in *file* which has been patched to gather profiling statistics on each basic block and dump them to *file***.prof** on exit. The report program can be used to generate various reports from this information. The **-X** option may be useful with this option.

-S *section*
Analyze *section* instead of text.

-X *routine*
Declare *routine* to be the name of a subroutine which causes the program to exit. When the **-P** option is used, this routine, when called, will dump the accumulated statistics to the **.prof** file. After writing the statistics data set to the **.prof** file, the statistics are reset to zero. When the **-O** option is used, the **-X** option will exclude the named routine from the optimization.

-W *routine*[=*weight*]
Specify a weighting factor for counting lis instructions in routine *routine*. If *weight* is omitted, it will default to 5. This option is used with the **-O** option.

-a *routine*
Add the specific named routine to the list of routines to be analyzed. This can

be used after **-N** to add a routine to the list. If used without **-N**, it assumes you meant to specify **-N**, and supplies one for you.

-d *file* Generate a detailed disassembly listing of each routine included in the analysis. The listing is done on a per basic block basis. By default this only generates the assembler listing, the clock cycle each instruction executes at (relative to the beginning of each basic block), and the reason any instruction is delayed. Use the **-v** option for more detail. Use the **-Zstage_status** option for much more verbose status of each pipeline stage each cycle.

-g *file* Generate global program statistics to file.

-i Print various informative bits of information about the object file.

-n Use nesting level to weight the count of `lis` instructions. This option is used with the **-O** option.

-r *file* Print summary statistics for each routine to file.

-s *routine*

Subtract a routine from the list to be analyzed. It pairs with the **-A** option much like **-N** and **-a** team up, only inverted.

-v Annotate the disassembly listing with the details about which instructions are using which machine resources at each cycle.

-w Suppress the output of warning messages.

-Z *keyword*

Pass a keyword option to **analyze**. The keywords recognized on the **-Z** option are:

PPC604 Disassemble instructions as they would be interpreted on a PowerPC 604 system. By default, instructions are disassembled as they would be interpreted relative to a PowerPC 604 system. It also causes **-C** to emulate the cache behavior of the PowerPC 604 system.

PPC601 Disassemble instructions as they would be interpreted on a PowerPC 601 system. By default, instructions are disassembled as they would be interpreted relative to a PowerPC 601 system. It also causes **-C** to emulate the cache behavior of the PowerPC 601 system.

break=*name*

Tell **analyze** the name of the global variable used to contain the break address. This variable is used by the `brk()` and `sbrk()` routines to track the next available heap address. When using the **-P** option, the initial value of this variable must be patched. The default name is `curbrk`.

exclude=register

Exclude the named register from the list of registers used to optimize out `lis` instruction. It may be used multiple times to exclude more than one register. Normally the **-O** option uses registers `r28` though `r31`.

help Give a short list and description of keyword options.

include=*register*
Add the named register to the list of registers used to optimize out lis instructions. It may be used multiple times to include more than one register. Normally the **-O** option uses registers `r28` though `r31`. However, if no routine in a program uses r6 though `r27` or the frame pointer, `r2`, these registers can be used too. **Analyze** will exit with an error if it finds a use of any of the named registers.

l2cache=*cache_size*[,*block_size*]
Define the characteristics of the secondary (L2) cache for use with the **-C** option. The *cache_size* argument is the total secondary cache size. It may be suffixed with **M** for megabytes or **K** for kilobytes. A *cache_size* of 0 means that there is no secondary cache. The optional *block_size* argument is the cache block (line) size in bytes; it defaults to 64 bytes. For example, **-Zl2cache=1m,128** specifies a secondary cache size of 1 megabyte with 128 bytes per cache block. If this option is not used, the secondary cache is 1 megabyte with 64-byte cache blocks. (Note that the first character is the letter `l` not the number `1`.)

options=*filename*
Tell **analyze** to read *filename* for a list of additional options. Each additional option should be on a separate line.

pdcache=*cache_size*[,*block_size*[,*sets*]]
Define the characteristics of the primary data cache for use with the **-C** option. The *cache_size* argument is the total primary data cache size. It may be suffixed with **M** for megabytes or **K** for kilobytes. A *cache_size* of 0 is not permitted. The optional *block_size* argument is the cache block (line) size in bytes; it defaults to 64 bytes. The optional *sets* argument is the number of sets; it defaults to *cache_size* divided by *block_size*. For example,**-Zpdcache=32k,32,128** specifies an 8-way associative primary data cache of size 32768 bytes with 128 sets, each set containing 8 cache blocks 32 bytes long.

This option also indicates that the primary data cache is separate from the primary instruction cache; therefore, it may not be used with the **-Zpucache** option. If this option is not used, the data cache characteristics are determined by the CPU type.

picache=*cache_size*[,*block_size*[,*sets*]]
Define the characteristics of the primary instruction cache for use with the **-C** option. The *cache_size* argument is the total primary instruction cache size. It may be suffixed with **M** for megabytes or **K** for kilobytes. A *cache_size* of 0 is not permitted. The optional *block_size* argument is the cache block (line) size in bytes; it defaults to 64 bytes. The optional *sets* argument is the number of sets; it defaults to *cache_size* divided by *block_size.* For example, **-Zpicache=32k,32,128** specifies an 8-way associative primary instruction cache of size 32768 bytes with 128 sets, each

set containing 8 cache blocks 32 bytes long.

This option also indicates that the primary instruction cache is separate from the primary data cache; therefore, it may not be used with the **-Zpucache** option. If this option is not used, the instruction cache characteristics are determined by the CPU type.

pucache=*cache_size*[,*block_size*[,*sets*]]
Define the characteristics of the unified primary cache for use with the **-C** option. The *cache_size* argument is the total unified primary cache size. It may be suffixed with **M** for megabytes or **K** for kilobytes. A *cache_size* of 0 is not permitted. The optional *block_size* argument is the cache block (line) size in bytes; it defaults to 64 bytes. The optional *sets* argument is the number of sets; it defaults to *cache_size* divided by *block_size*. For example, **-Zpucache=32k,64,64** specifies an 8-way associative unified primary cache of size 32768 bytes with 64 sets, each set containing 8 cache blocks 64 bytes long.

This option also indicates that a single primary cache is used for both instructions and data; therefore, it may not be used with the **-Zpdcache** or **-Zpicache** options. If this option is not used, the cache characteristics are determined by the CPU type.

retain Retain the extra relation information that the Concurrent link editor to the object file. This information is provided so **analyze** can optimize things like assigned gotos correctly. Normally this information is stripped after optimization. If you are going to want to profile or disassemble the program file, this option will retain the extra relocation information so the additional processing can be more accurate.

rmask=*register_list*
Specify a list of registers to be considered live at a return instruction. To optimize pure C or Fortran 77 code, use **-Zrmask=r3r4**. The default mask contains `r3` through `r15`.

rtag=*character*
Specify a character to enclose all routine names output in reports. This is for an Ada filter that translates raw routine names into Ada R.M. expanded names.

stag=*character*
Specify a character to enclose all source file names output in reports. This is for an Ada filter that translates raw source file names into actual file names.

stage_status
Add output describing the status of all pipeline stages each cycle to the disassembly output of the **-d** option.

strip Strip the object file before writing it out.

Assumptions and Constraints

The timing information is not totally accurate. The worst-case timing information should really be generated by propagating live on entry resource utilizations backwards through the flow graph to see how they interact with live on exit resource utilizations from the predecessor blocks, but this is complex and would require a great deal more code to do the analysis.

`analyze` assumes all memory references are cache hits. Thus, the timing information assumes there will never be any cache misses or memory wait states since a static analysis cannot know if a memory reference will be in the cache or not. Note that the **`-C`** option does not circumvent this restriction.

The **`-C`** option cannot provide a completely accurate model of the real cache because the simulation is not aware of other running processes nor of the operating system itself. The purpose of this option is to provide a measure of the locality of a user program.

With the **`-C`** option, loads and stores that access multiple storage locations (e.g., `lmw` or `stmw`) are treated as if they access only the first unit of storage. Also, if multiple consecutive accesses of a cache block occur, only one miss is recorded. In reality, multiple misses might occur while the cache block is loading. Finally, conditional stores (`stwcx.`) are assumed to always succeed.

For more detailed information on the hardware architectures, refer to the following publications:

PowerPC User Instruction Set Architecture

PowerPC 604 User's Manual

report

The **`report`** tool reads the profile data generated by running a program which has been patched with the **`-P`** option of the **`analyze`** tool.

`report` needs two arguments, the name of the patched program (generated by **`analyze`**) and the name of the profile data file. If the second argument is not specified, it will append to the end of the first argument and look there for the profile data.

The printed reports are generated in a format that will conform with tools that are used to parse error messages from compilers, with

file name `:` *line number*

listed first on the line.

Except where explicitly indicated in the individual report, all times are reported in terms of cycles. Because no analysis of pipe conflicts across basic blocks is done, times are always reported in terms of a range of times from max time to min time. All sorts are done on max time, and percentages are calculated in terms of max time.

Usage

`report` is invoked as follows:

> **`report [-H] [-a] [-b] [-B`** *number***`] [-c] [-d`** *range***`] [-i]`**
> **`[-l] [-m] [-M`** *megahertz***`] [-n] [-N`** *number***`] [-o] [-O`** *number***`]`**
> **`[-r] [-R`** *number***`] [-s] [-t] [-T`** *file***`] [-w] [-Y`** *character***`] [-z]`**
> **`[-Z`** *character***`]`** *programfile* **[***programfile.prof***]**

The *programfile* argument specifies the name of the executable file over which **`analyze`** has patched, for producing profile information. All other arguments are optional and are as follows:

`-H` Print a help message and exit.

`-a` Report on all the individual data sets recorded in the profile data file. If the **`-t`** option is used, normally only the totals for all the runs of the program are printed. The **`-a`** option forces all the individual runs to generate reports as well.

`-b` Generate a report showing where the program spent its time at the individual basic block level. This report is ordered with the most expensive block listed first (in terms of cycles spent in that block).

`-B` *number*
Restrict the basic block report to only the first *number* basic blocks. If the number is written with a trailing `%` character on the end, then it will restrict the list of blocks printed to just the first set of blocks that total to that percent of the total time. This option implies the use of the **`-b`** option. Both forms of the **`-B`** option may be used, in which case the first limit reached will terminate the listing.

`-c` List the names of routines called by each routine.

`-d` *range* Select which data sets to report. Each time a patched program is run, it appends a new set of profile data onto the end of its profile data file. This means that one data file may contain several sets of data. The first set is set number one. This option may be used to select which sets are actually reported. *Range* can be a single number, a list of comma separated numbers or a range of numbers separated with a dash.

`-i` Report summary information for the complete program. This option uses the assumed clock rate (specified with the **`-M`** option) to report the clock time the original program would take to run. It also summarizes the count of the different types of instructions that were executed.

`-l` Use max time instead of min time when sorting statistics and computing percentages.

`-m` Print timing information in milliseconds rather than cycles.

`-M` *megahertz*
Specify the megahertz clock rate to assume when computing clock time from cycle counts. The default is 100.

-n Generate a profiling report showing the number of cache misses due to data accesses (loads and stores). The report is sorted in decreasing order of secondary cache misses.

-N *number*
Limit the number of data access cache statistics printed. Use of **-N** implies **-n**. This option acts much like the **-B** option (above).

-o Generate a profiling report showing the number of cache misses due to instruction fetches. The report is sorted in decreasing order of secondary cache misses.

-O *number*
Limit the number of instruction access cache statistics printed. Use of **-O** implies **-o**. This option acts much like the **-B** option (above).

-r Generate a profiling report showing the time spent in each routine. This report is generated by adding up all the time in the individual basic blocks.

-R *number*
Restrict the routine report to only the first *number* routines. This option acts much like the **-B** option (above), and will accept an absolute number or a percentage. Use of **-R** implies **-r**.

-s Print the header information from each profile data set. This may be used by itself to determine how many sets of data are in a profile data file in order to determine which sets to examine with the **-d** option.

-t Total all the data sets and print the total statistics in any of the reports requested. Normally this option suppresses the generation of any reports on individual data sets and only the totals are printed. The **-a** option (above) can be used to change this behavior.

-T *file* Print a summary of all data sets to the specified file.

-w Print the raw statistics information from the profile data file in a human readable form.

-Y *character*
Specify a character which is used to enclose all routine names output in reports. This is for an Ada filter that translates raw routine names into Ada R.M. expanded names.

-z Print information about blocks and routines that are executed zero times. If no blocks in a routine were executed, only the routine name is printed and the individual blocks for that routine are not reported. For routines in which some blocks were executed and some were not, the blocks with zero execution time are reported individually.

-Z *character*
Specify a character which is used to enclose all source file names output in reports. This is for an Ada filter that translates raw source file names into actual file names.

Assumptions and Constraints

The cycle counts reported are based on the cycle counts calculated by **`analyze`** and are subject to the same limitations described in the documentation for that tool.

Most notably the behavior of the memory system is not taken into account, so actual wall time may be even longer than the maximum time reported (especially if the application has many cache misses).

If the program was not compiled with line number information, or if it was stripped before being processed by **`analyze`**, none of the reports will be able to include accurate file names or line numbers. (Generally the file name will be the null string, and the line numbers will be `0` if the information was not present in the object file).

3
Project Control

Replace with Part 3 tab

Part 3 - Project Control

12
Introduction to Project Control

12 Introduction to Project Control

Introduction

A *software project* consists of one or more products. Each *product* consists of one or more files, including the following:

- Program files, for example, source, object, and executables for one or more platforms
- Documentation files, for example, design and functional specifications, release notes, man pages, user and reference manuals, and reference cards
- Training files, for example, student guides, instructor guides, and example source files
- Testing files, for example, third-party and internally developed test suites and programs supplied with error reports

You can save time by using tools to automate project management. This part of the manual covers tools that give you control over projects, products, and files. For example:

- Remembering file locations and dependencies and product-generation steps for a developing product can be cumbersome. You can store this information in description files for the **`make`** tool to process. Chapter 13 ("Managing File Interactions with make") discusses **`make`** and its description files.
- Retaining an audit trail of editing changes can be useful in debugging and documenting a developing product. Chapter 14 ("Tracking Versions with SCCS") describes SCCS, the Source Code Control System, that allows you to capture this information.

13
Managing File Interactions with make

13 Managing File Interactions with make

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

`make` provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget

- File-to-file dependencies
- Files that were modified and the impact that has on other files
- The exact sequence of operations needed to generate a new version of the program

`make` keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **`make`** command creates the finished program by recompiling only those portions directly or indirectly affected by the change. The relationships between files and the processes that generate files are specified by the user in a description file.

The basic operation of **`make`** is to

- Find the target in the description file
- Ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- (Re)create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on inter-file dependencies and command sequences is conventionally called **`makefile`**, **`Makefile`**, **`s.makefile`**, or **`s.Makefile`**. If this naming convention is followed, the simple command **`make`** is usually sufficient to regenerate the target regardless of the number of files edited since the last **`make`**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than entering all the commands to regenerate the target, entering the **`make`** command ensures that the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program builds and searches a graph of these dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- A user-supplied description file
- File names and last-modified times from the file system
- Built-in rules supply default dependency information and implied commands

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the math library, **libm**. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command

```
make
```

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to combine them to create **prog.** The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (that is, issue a **cc -c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to relink **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see “Description Files and Substitutions” on page 13-6.) Thus, an entry `save` might be included to copy a certain set of files, or an entry `clean` might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by the symbol = followed by what the macro stands for. A macro is invoked by preceding the name by the symbol `$`. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

`$*`, `$@, $?, and $<` are four special macros that change values during the execution of the command. (These four macros are described later in “Description Files and Substitutions” on page 13-6.) The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o  y.o  z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

```
make  LIBES="-ll -lm"
```

loads the three objects with both the **lex** (**-ll**) and the math (**-lm**) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX system commands, arguments with embedded blanks must somehow be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c \
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o \
          dosys.o gram.o
LIBES =
LINT = lint -p
CFLAGS = -O
LP = lp

make:  $(OBJECTS)
        $(CC) $(CFLAGS) -o make $(OBJECTS) $(LIBES)
        @size make

$(OBJECTS):  defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        make
        @size make /usr/ccs/bin/make
        cp make /usr/ccs/bin/make && rm make

lint:    dosys.c doname.c files.c main.c misc.c gram.c
         $(LINT) dosys.c doname.c files.c main.c misc.c \
         gram.c

             # print files that are out-of-date
             # with respect to "print" file.

print:   $(FILES)
         pr $? | $(LP)
         touch print
```

The **make** program prints out each command before issuing it.

The following output results from entering the command **make** in a directory containing only the source and description files:

```
  cc  -O -c main.c
  cc  -O -c doname.c
  cc  -O -c misc.c
  cc  -O -c files.c
  cc  -O -c dosys.c
  yacc  gram.y
  mv y.tab.c gram.c
  cc  -O -c gram.c
  cc  -o make  main.o doname.o misc.o files.o dosys.o gram.o
13188 + 3348 + 3044 = 19580
```

The last line results from the **`size make`** command. The printing of the command line itself was suppressed by the symbol `@` in the description file.

Parallel make

If **`make`** is invoked with the **`-P`** option, it tries to build more than one target at a time, in parallel. (This is done by using the standard UNIX system process mechanism which enables multiple processes to run simultaneously.)

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

For the **`makefile`** shown above, it would create processes to build **`x.o`**, **`y.o`** and **`z.o`** in parallel. After these processes were complete, it would build **`prog`**.

The number of targets **`make`** will try to build in parallel is determined by the value of the environment variable `PARALLEL`. If **`-P`** is invoked, but `PARALLEL` is not set, then **`make`** will try to build no more than two targets in parallel.

You can use the `.MUTEX` directive to serialize the updating of some specified targets. This is useful when two or more targets modify a common output file, such as when inserting modules into an archive or when creating an intermediate file with the same name, as is done by **`lex`** and **`yacc`**.

If the **`makefile`** above contained a `.MUTEX` directive of the form

```
.MUTEX: x.o y.o
```

it would prevent **`make`** from building **`x.o`** and **`y.o`** in parallel.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that the symbol # and all characters on the same line after it are ignored. Blank lines and lines beginning with # are totally ignored.

Continuation Lines

If a non-comment line is too long, the line can be continued by using the symbol `\`, which must be the last character on the line. If the last character of a line is `\`, then it, the new-line, and all following blanks and tabs are replaced by a single blank. Comments can be continued on to the next line as well.

Macro Definitions

A macro definition is an identifier followed by the symbol `=`. The identifier must not be preceded by a colon (`:`) or a tab. The name (string of letters and digits) to the left of the = (trailing blanks and tabs are stripped) is assigned the string of characters following the = (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns `LIBES` the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **`make`**'s own rules.

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
...
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as `*` and `?` are expanded when the commands are evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab (denoted above as `\t`) immediately following a dependency line. A command is any string of characters not including `#`, except when # is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in "Archive Libraries" on page 13-11.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (**`-s`** option of the **`make`** command) or if the command line in the description file begins with an `@` sign. **`make`** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **`-i`** flag has been specified on the **`make`** command line, if the fake target name `.IGNORE` appears in the description file, or if the command string in the description file begins with a hyphen (`-`). If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (**`cd`** and shell control commands, for instance) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The `$@` macro is set to the full target name of the current target. The `$@` macro is evaluated only for explicitly named dependencies. The `$?` macro is set to the string of names that were found to be younger than the target. The `$?` macro is evaluated when explicit rules from the **`makefile`** are evaluated. If the command was generated by an implicit rule, the `$<` macro is the name of the related file that caused the action; and the `$*` macro is the prefix shared by the current and the dependent file names. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name `.DEFAULT` are used. If there is no such name, **`make`** prints a message and stops.

In addition, a description file may also use the following related macros: `$(@D)`, `$(@F), $(*D), $(*F), $(<D), and $(<F)` (see below).

Extensions of $*, $@, and $<

The internally generated macros `$*`, `$@`, and `$<` are useful generic terms for current targets and out-of-date relatives. To this list is added the following related macros: `$(@D), $(@F), $(*D), $(*F), $(<D), and $(<F).`The `D` refers to the directory part of the single character macro. The `F` refers to the file name part of the single character macro. These additions are useful when building hierarchical **`makefile`**s. They allow access to directory names for purposes of using the **`cd`** command of the shell. Thus, a command can be

```
cd $(<D); $(MAKE) $(<F)
```

Output Translations

The values of macros are replaced when evaluated. The general form, where brackets indicate that the enclosed sequence is optional, is as follows:

`$`**`(`***macro***`[:`***string1***`=[`***string2***`]])`**

The parentheses are optional if there is no substitution specification and the macro name is a single character. If a substitution sequence is present, the value of the macro is considered to be a sequence of “words” separated by sequences of blanks, tabs, and new-line characters. Then, for each such word that ends with *string1*, *string1* is replaced with *string2* (or no characters if *string2* is not present).

This particular substitution capability was chosen because **`make`** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (that is, files ending in **`.c`**). Thus, the following fragment optimizes the executions of **`make`** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **`make`** generates.

Recursive Makefiles

Another feature of **`make`** concerns the environment and recursive invocations. If the sequence `$(MAKE)` appears anywhere in a shell command line, the line is executed even if the **`-n`** flag is set. Since the **`-n`** flag is exported across invocations of **`make`** (through the `MAKEFLAGS` variable), the only thing that is executed is the **`make`** command itself. This feature is useful when a hierarchy of **`makefile`**s describes a set of software subsystems.

For testing purposes, **`make -n`** can be executed and everything that would have been done will be printed including output from lower-level invocations of **`make`**.

Suffixes and Transformation Rules

`make` uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **`-r`** flag is used on the **`make`** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name `.SUFFIXES.` **`make`** searches for a file with any of the suffixes on the list. If it finds one, **`make`** transforms it into a file with another suffix. Transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **`.r`** file to a **`.o`** file is thus **`.r.o`**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **`.r.o`** is used. If a command is generated by using one of these suffixing rules, the macro `$*` is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro `$<` is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for `.SUFFIXES` in the description file. The dependents are added to the usual list. A `.SUFFIXES` line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

`make` uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list (in order) is as follows:

`.o`	Object file
`.c`	C source file
`.c~`	SCCS C source file
`.y`	**`yacc`** C source grammar
`.y~`	SCCS **`yacc`** C source grammar
`.l`	**`lex`** C source grammar
`.l~`	SCCS **`lex`** C source grammar
`.s`	Assembler source file
`.s~`	SCCS assembler source file
`.sh`	Shell file

.sh~	SCCS shell file
.h	Header file
.h~	SCCS header file
.f	Fortran source file
.f~	SCCS Fortran source file
.C	C++ source file
.C~	SCCS C++ source file
.Y	yacc C++ source grammar
.Y~	SCCS yacc C++ source grammar
.L	lex C++ source grammar
.L~	SCCS lex C++ source grammar

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

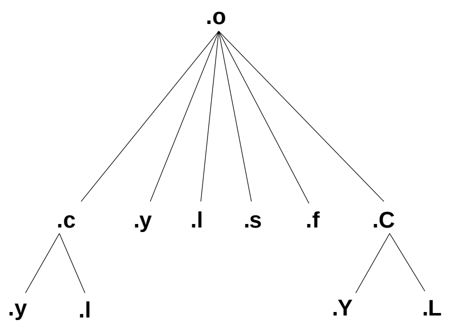


Figure 13-1. Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 13-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros `AS`, `CC`, `C++C`, `F77`, `YACC`, and `LEX`. The command

```
make CC=newcc
```

will cause the **newcc** command to be used instead of the usual C language compiler. The macros CFLAGS, YFLAGS, LFLAGS, ASFLAGS, FFLAGS, and C++FLAGS may be set to cause these commands to be issued with optional flags. Thus

```
make CFLAGS=-g
```

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

```
projlib(object.o)
```

or

```
projlib((entry_pt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
        $(CC) -c $(CFLAGS) pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::   projlib(pfile2.o)
        $(CC) -c $(CFLAGS) pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

and so on for each object. This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** file. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde (~) syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
            @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $(<F:.c=.o)
        rm -f $(<F:.c=.o)
```

Thus, the `$@` macro is the **`.a`** target (`projlib`); the `$<` and `$*` macros are set to the out-of-date C language file, and the file name minus the suffix, respectively (**`pfile1.c`** and **`pfile1`**). The `$<` macro (in the preceding rule) could have been changed to `$*.c`.

It is useful to go into some detail about exactly what **`make`** does when it sees the construction

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **`pfile1.c`**. Also, there is no **`pfile1.o`** file.

1. **`make`** `projlib`.
2. Before **`make`**ing `projlib`, check each dependent of `projlib`.
3. `projlib(`**`pfile1.o`**`)` is a dependent of `projlib` and needs to be generated.
4. Before generating `projlib(`**`pfile1.o`**`)`, check each dependent of `projlib(`**`pfile1.o`**`)`. (There are none.)
5. Use internal rules to try to create `projlib(`**`pfile1.o`**`)`. (There is no explicit rule.) Note that `projlib(`**`pfile1.o`**`)` has a parenthesis in the name to identify the target suffix as **`.a`**. This is the key. There is no explicit **`.a`** at the end of the `projlib` library name. The parenthesis implies the **`.a`** suffix. In this sense, the **`.a`** is hard-wired into **`make`**.
6. Break the name `projlib(`**`pfile1.o`**`)` up into `projlib` and **`pfile1.o`**. Define two macros, `$@` (`projlib`) and `$*` (**`pfile1`**).
7. Look for a rule **`.X.a`** and a file **`$*.X`**. The first **`.X`** (in the `.SUFFIXES` list) which fulfills these conditions is **`.c`** so the rule is **`.c.a`**, and the file is **`pfile1.c`**. Set `$<` to be **`pfile1.c`** and execute the rule. In fact, **`make`** must then compile **`pfile1.c`**.
8. The library has been updated. Execute the command associated with the **`projlib:`** dependency, namely

   ```
   @echo projlib up-to-date
   ```

It should be noted that to let **`pfile1.o`** have dependencies, the following syntax is required:

```
projlib(pfile1.o):        $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The `$%` macro is evaluated each time `$@` is evaluated. If there is no current archive member, `$%` is null. If an archive member exists, then `$%` evaluates to the expression between the parenthesis.

Source Code Control System File Names

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. SCCS files are the exception. Here, **s.** precedes the file name part of the complete path name.

To allow **make** easy access to the prefix **s.** the symbol ~ is used as an identifier of SCCS files. Hence, `.c~.o` refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        rm -f $*.c
```

Thus, ~ appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) **~.**

The following SCCS suffixes are internally defined:

```
.c~         .sh~        .C~
.y~         .h~         .Y~
.l~         .f~         .L~
.s~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:        .s~.s:      .C~:
.c~.c:      .s~.a:      .C~.C:
.c~.a:      .s~.o:      .C~.a:
.c~.o:      .sh~:       .C~.o:
.y~.c:      .sh~.sh:    .Y~.C:
.y~.o:      .h~.h:      .Y~.o:
.y~.y:      .f~:        .Y~.Y:
.l~.c:      .f~.f:      .L~.C:
.l~.o:      .f~.a:      .L~.o:
.l~.l:      .f~.o:      .L~.L:
.s~:
```

Obviously, the user can define other rules and suffixes that may prove useful. The ~ provides a handle on the SCCS file name format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the UNIX system program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
        $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)
```

In fact, this `.c:` rule is internally defined so no **`makefile`** is necessary at all. The user only needs to enter

```
make cat dd echo date
```

(these are all UNIX system single-file programs) and all four C language source files are passed through the above shell command line associated with the `.c:` rule. The internally defined single suffix rules are

```
.c:         .sh:        .f~:
.c~:        .sh~:       .C:
.s:         .f:         .C~:
.s~:
.sh:
```

Others may be added in the **`makefile`** by the user.

Included Files

The **`make`** program has a capability similar to the `#include` directive of the C preprocessor. If the string `include` appears as the first seven letters of a line in a **`makefile`** and is followed by a blank or a tab, the rest of the line is assumed to be a file name, which the current invocation of **`make`** will read. Macros may be used in file names. The file descriptors are stacked for reading `include` files so that no more than 16 levels of nested `includes` are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **`make`**. That is, if **`make`** is typed and only a file named **`s.makefile`** or **`s.Makefile`** exists, **`make`** will do a **`get`** on the file, then read and remove the file.

Dynamic Dependency Parameters

A dynamic dependency parameter has meaning only on the dependency line in a **`makefile`**. The `$$@` refers to the current “thing” to the left of the `:` symbol (which is `$@`). Also the form `$$(@F)` exists, which allows access to the file part of `$@`. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **`cat.c`**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX system software command directory could have a **`makefile`** like:

```
CMDS = cat dd echo date cmp comm chown
```

```
$(CMDS):        $$@.c
       $(CC) $(CFLAGS) $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **`makefile`** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **`makefile`**.

The second useful form of the dependency parameter is `$$(@F)`. It represents the file name part of `$$@`. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **`/usr/include`** directory from **`makefile`** in the **`/usr/src/head`** directory. Thus, the **`/usr/src/head/makefile`** would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **`/usr/include`** directory whenever one of the above files in **`/usr/src/head`** was updated.

Viewpaths (VPATH)

This implementation of **`make(1)`** has been enhanced to support `VPATH` functionality or the concept of viewpaths. `VPATH` is a macro that allows one to specify a list of directories to search for the files **`make(1)`** needs to complete its tasks. The viewpath may be specified in one or more of four methods. First, it may be specified on the command line with the **`-v`** *viewpath* option; where *viewpath* is some directory (absolute or relative to the current working directory) other than the current working directory. Second, it may be specified on the command line in the `VPATH=` macro specification as a colon separated list of directories to be searched. The third method is to specify the `VPATH=` macro within the **`makefile`** being used; again, as a blank- or colon-separated list of directories. Finally, the `VPATH` may be specified within the environment by setting the `VPATH=` environment variable to a blank- or colon-separated list of directories. In all cases, the directories specified may be absolute paths or relative to the current working directory. The methods have been identified in the order of precedence; in other words, the method of using **`-v`** *viewpath* takes precedence over the others.

Examples of how to invoke use these methods are illustrated below:

To search for components in the current working directory and **`/usr/src`**:

```
make -f makefile -v /usr/src
```

To search for components in the current working directory, **`mysrc`**, and **`/usr/src`** in that order:

```
make -f makefile VPATH=mysrc:/usr/src
```

To search for components in the current working directory, **`mysrc`**, **`/usr/src`**, and **`yoursrc`** in that order:

```
VPATH = mysrc:/usr/src:${DIR1}
DIR1 = yoursrc
OBJS = main.o allocate.o delete.o
outfile: ${OBJS}
         ${CC} -o $@ ${OBJS}
main.o: main.c
allocate.o: allocate.c
delete.o: delete.c
```

With this enhancement, SCCS directories can now be searched for build components simply by specifying the SCCS directory in one of the above methods.

This enhancement also allows for the expansion of the `VPATH` as new makefiles are included or referenced through the initial invocation.

Some limitations on the `VPATH` include: any one path specified cannot be longer than `MAXPATHLEN-1` and the maximum number of paths specified, regardless of length, cannot exceed 10.

Command Usage

Refer to **`make(1)`** for detailed information.

The make Command

The **`make`** command takes macro definitions, options, description file names, and target file names as arguments in the form:

`make [-f` *makefile***`] [-v`** *viewpath***`] [-eiknpPqrstuw] [`***names***`]`**

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded = symbols) are analyzed and the assignments made. Command line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

`-i` Ignore error codes returned by invoked commands. This mode is entered if the fake target name `.IGNORE` appears in the description file.

-s	Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name `.SILENT` appears in the description file.
-r	Do not use the built-in rules.
-n	No execute mode. Print commands, but do not execute them. Even lines beginning with an `@` sign are printed.
-t	Touch the target files (causing them to be up to date) rather than issue the usual commands.
-q	Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.
-p	Print out the complete set of macro definitions and target descriptions.
-k	Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.
-e	Environment variables override assignments within **makefile**s.
-f *makefile*	Description file name. *makefile* is assumed to be the name of a description file. A *makefile* of `-` denotes the standard input. If there are no **-f** arguments, the file named **makefile, Makefile, s.makefile,** or **s.Makefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.
-P	Update, in parallel, more than one target at a time. The number of targets updated concurrently is determined by the environment variable `PARALLEL` and the presence of `.MUTEX` directives in makefiles.
-u	Unconditionally make the target, ignoring all timestamps.
-v *viewpath*	Absolute or relative path name (*viewpath*) to be searched for needed files.
-w	Suppress warning messages. Fatal messages will not be affected.

The following fake target names are evaluated in the same manner as flags:

`.DEFAULT`	If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name `.DEFAULT` are used if it exists.
`.PRECIOUS`	Dependents on this target are not removed when quit or interrupt is pressed.
`.SILENT`	Same effect as the **-s** option.
`.IGNORE`	Same effect as the **-i** option.

`.PRECIOUS`	Dependents of the `.PRECIOUS` entry will not be removed when quit or interrupt are pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with the symbol **`.`** is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **`make`** executes. Precedence is a prime consideration in doing this properly. The following describes **`make`**'s interaction with the environment. A macro, `MAKEFLAGS,` is maintained by **`make`**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to recursive invocations of **`make`**. Command line flags and assignments in the **`makefile`** update `MAKEFLAGS.` Thus, to describe how the environment interacts with **`make`**, the `MAKEFLAGS` macro (environment variable) must be considered.

When executed, **`make`** assigns macro definitions in the following order:

1. Read the `MAKEFLAGS` environment variable. If it is not present or null, the internal **`make`** variable `MAKEFLAGS` is set to the null string. Otherwise, each letter in `MAKEFLAGS` is assumed to be an input flag argument and is processed as such. (The only exceptions are the **`-f`**, **`-p`**, and **`-r`** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense).
4. Read the **`makefile`**(s). The assignments in the **`makefile`**(s) override the environment. This order is chosen so that when a **`makefile`** is read and executed, you know what to expect. That is, you get what is seen unless the **`-e`** flag is used. The **`-e`** is the input flag argument, which tells **`make`** to have the environment override the **`makefile`** assignments. Thus, if **`make -e`** is entered, the variables in the environment override the definitions in the **`makefile`**. Also `MAKEFLAGS` overrides the environment if assigned. This is useful for further invocations of **`make`** from the current **`makefile`**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. Internal definitions
2. Environment
3. **`makefile`**(s)
4. Command line

The **`-e`** flag has the effect of rearranging the order to:

1. Internal definitions
2. **`makefile`**(s)

3. Environment
4. Command line

This order is general enough to allow a programmer to define a **`makefile`** or set of **`makefile`**s whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **`make`**'s specific meaning of dependency. If file **`x.c`** has a

```
#include "defs.h"
```

line, then the object file **`x.o`** depends on **`defs.h`**; the source file **`x.c`** does not. If **`defs.h`** is changed, nothing is done to the file **`x.c`** while file **`x.o`** must be recreated.

To discover what **`make`** would do, the **`-n`** option is very useful. The command

```
make -n
```

orders **`make`** to print out the commands that **`make`** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (adding a comment to an **`include`** file, for example), the **`-t`** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **`make`** updates the modification times on the affected file. Thus, the command

```
make -ts
```

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **`make`** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **`make`** are reproduced below.

```
#
#   SUFFIXES RECOGNIZED BY MAKE
#

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .f .f~ .C .C~ \
           .Y .Y~ .L .L~

#
#   PREDEFINED MACROS
#

AR=ar
ARFLAGS=rv
AS=as
ASFLAGS=
BUILD=build
CC=cc
CFLAGS=-O
C++C=CC
C++FLAGS=-O
F77=f77
FFLAGS=-O
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
MAKE=make
YACC=yacc
YFLAGS=
#
#   SPECIAL RULES
#

markfile.o : markfile
    A=@; echo "static char _sccsid[]=\042`grep $$A'(#)' markfile`\042;" \
    > markfile.c
    $(CC) -c markfile.c
    -rm -f markfile.c
#
#   SINGLE SUFFIX RULES
#

.c:
    $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)

.c~:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) -o $@ $*.c $(LDFLAGS)
    -rm -f $*.c
```

Screen 13-1. make Internal Rules

```
.s:
    $(AS) $(ASFLAGS) -o $*.o $<
    $(CC) -o $@ $*.o $(LDFLAGS)
    -rm -f $*.o

.s~:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    $(CC) -o $* $*.o $(LDFLAGS)
    -rm -f $*.[so]

.sh:
    cp $< $@; chmod +x $@

.sh~:
    $(GET) $(GFLAGS) $<
    cp $*.sh $*; chmod +x $@
    -rm -f $*.sh

.f:
    $(F77) $(FFLAGS) -o $@ $< $(LDFLAGS)

.f~:
    $(GET) $(GFLAGS) $<
    $(F77) $(FFLAGS) -o $@ $*.f $(LDFLAGS)
    -rm -f $*.f

.C:
    $(C++C) $(C++FLAGS) -o $@ $< $(LDFLAGS)

.C~:
    $(GET) $(GFLAGS) $<
    $(C++C) $(C++FLAGS) -o $@ $*.C $(LDFLAGS)
    -rm -f $*.C
#
#   DOUBLE SUFFIX RULES
#

.c~.c .y~.y .l~.l .s~.s .sh~.sh .h~.h: .f~.f .C~.C .Y~.Y .L~.L:
    $(GET) $(GFLAGS) $<

.c.a:
    $(CC) $(CFLAGS) -c $<
    $(AR) $(ARFLAGS) $@ $(<F:.c=.o)
    -rm -f $(<F:.c=.o)

.c~.a:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) -c $*.c
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[co]
```

```
.c.o:
    $(CC) $(CFLAGS) -c $<

.c~.o:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c

.y.c:
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@

.y~.c:
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.y
    mv y.tab.c $*.c
    -rm -f $*.y

.y.o:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    -rm -f y.tab.c
    mv y.tab.o $@

.y~.o:
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.y
    $(CC) $(CFLAGS) -c y.tab.c
    -rm -f y.tab.c $*.y
    mv y.tab.o $*.o

.l.c:
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@

.l~.c:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    mv lex.yy.c $@
    -rm -f $*.l

.l.o:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    -rm -f lex.yy.c
    mv lex.yy.o $@

.l~.o:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    $(CC) $(CFLAGS) -c lex.yy.c
    -rm -f lex.yy.c $*.l
    mv lex.yy.o $@
```

```
.s.a:
    $(AS) $(ASFLAGS) -o $*.o $*.s
    $(AR) $(ARFLAGS) $@ $*.o

.s~.a:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[so]

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    -rm -f $*.s

.f.a:
    $(F77) $(FFLAGS) -c $*.f
    $(AR) $(ARFLAGS) $@ $(<F:.f=.o)
    -rm -f $(<F:.f=.o)

.f~.a:
    $(GET) $(GFLAGS) $<
    $(F77) $(FFLAGS) -c $*.f
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[fo]

.f.o:
    $(F77) $(FFLAGS) -c $*.f

.f~.o:
    $(GET) $(GFLAGS) $<
    $(F77) $(FFLAGS) -c $*.f
    -rm -f $*.f

.C.a:
    $(C++C) $(C++FLAGS) -c $<
    $(AR) $(ARFLAGS) $@ $(<F:.C=.o)
    -rm -f $(<F:.C=.o)

.C~.a:
    $(GET) $(GFLAGS) $<
    $(C++C) $(C++FLAGS) -c $*.C
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[Co]

.C.o:
    $(C++C) $(C++FLAGS) -c $<

.C~.o:
    $(GET) $(GFLAGS) $<
    $(C++C) $(C++FLAGS) -c $*.C
    -rm -f $*.C
```

```
.Y.C:
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@

.Y~.C:
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.Y
    mv y.tab.c $*.C
    -rm -f $*.Y

.Y.o:
    $(YACC) $(YFLAGS) $<
    $(C++C) $(C++FLAGS) -c y.tab.c
    -rm -f y.tab.c
    mv y.tab.o $@

.Y~.o:
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.Y
    $(C++C) $(C++FLAGS) -c y.tab.c
    -rm -f y.tab.c $*.Y
    mv y.tab.o $*.o

.L.C:
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@

.L~.C:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.L
    mv lex.yy.c $@
    -rm -f $*.L

.L.o:
    $(LEX) $(LFLAGS) $<
    $(C++C) $(C++FLAGS) -c lex.yy.c
    -rm -f lex.yy.c
    mv lex.yy.o $@

.L~.o:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.L
    $(C++C) $(C++FLAGS) -c lex.yy.c
    -rm -f lex.yy.c $*.L
    mv lex.yy.o $@
```

14
Tracking Versions with SCCS

14
Tracking Versions with SCCS

Introduction

The Source Code Control System, SCCS, is a set of programs that you can use to track evolving versions of files, ordinary text files as well as source files. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History information can be stored with each version: why the changes were made, who made them, and when they were made.

This chapter covers the following topics:

- The basics of creating, retrieving, and updating an SCCS file;
- Delta numbering: how versions of an SCCS file are named;
- SCCS command conventions: what rules apply to SCCS commands;
- SCCS commands: the 14 SCCS commands and their more useful arguments;
- SCCS files: protection, format, and auditing of SCCS files.

Basic Usage

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS IDentification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file, that is, its first delta, is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. More on delta numbering later. At this point, it is enough to know that by default SCCS assigns SIDs automatically.

Creating an SCCS File with admin

Suppose you have a file called **lang** that is simply a list of five programming language names:

```
C
PL/I
Fortran
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** (for administer) command. The following creates an SCCS file from the **lang** file:

```
admin -ilang s.lang
```

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **-i** key letter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later under the **get** command in “SCCS Commands” on page 14-8. In the following examples, this warning message is not shown although it may be issued.

Remove the **lang** file. It is no longer needed because it exists now under SCCS as **s.lang**.

```
rm lang
```

Retrieving a File with get

The command

```
get s.lang
```

retrieves the latest version of **s.lang** and prints

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text is placed in a new file called **`lang`**. That is, if you list the contents of your directory, you will see both **`lang`** and **`s.lang`**.

The **`get s.lang`** command creates **`lang`**, a file meant for viewing (read-only), not for making changes to. If you want to make changes to it, the **`-e`** (edit) option must be used. This is done as follows:

```
get -e s.lang
```

`get -e` causes SCCS to create **`lang`** for both reading and writing (editing). It also places certain information about **`lang`** in another new file, called **`p.lang`**, which is needed later by the **`delta`** command. Now if you list the contents of your directory, you will see **`s.lang`**, **`lang`**, and **`p.lang`**.

`get -e` prints the same messages as **`get`**, except that the SID for the first delta you will create also is issued:

```
1.1
new delta 1.2
5 lines
```

Change **`lang`** by adding two more programming languages:

```
SNOBOL
ADA
```

Recording Changes with delta

Next, use the **`delta`** command as follows:

```
delta s.lang
```

`delta` then prompts with

```
comments?
```

Your response should be an explanation of why the changes were made. For example,

```
added more languages
```

`delta` now reads the file **`p.lang`** and determines what changes you made to **`lang`**. It does this by doing its own **`get`** to retrieve the original version and applying the **`diff(1)`** command to the original version and the edited version. Next, **`delta`** stores the changes in **`s.lang`** and destroys the no longer needed **`p.lang`** and **`lang`** files.

When this process is complete, **`delta`** outputs

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to **`s.lang`**.

More on get

The command

```
get s.lang
```

retrieves the latest version of the file **s.lang**, now 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, any of the following will retrieve version 1.2:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following **-r** are SIDs. When you omit the level number of the SID (as in **get -r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version the **delta** command will create. If the file is now edited — for example, by deleting `COBOL` from the list of languages — and **delta** is executed

```
delta s.lang
comments? deleted cobol from list of languages
```

you will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release can be created in a similar manner. A delta can still be made to the “old” release 1. This will be explained later in the chapter.

The help Command

If the command

```
get lang
```

is now executed, the following message will be output:

```
ERROR [lang]: not an SCCS file (co1)
```

The code `co1` can be used with **`help`** to print a fuller explanation of the message:

```
help co1
```

This gives the following explanation of why **`get lang`** produced an error message:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

`help` is useful whenever there is doubt about the meaning of almost any SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root node is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, etc. The components of these SIDs are called release and level numbers, respectively. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 14-1.

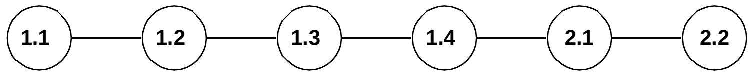


Figure 14-1. Evolution of an SCCS File

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 14-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (that is, deltas 1.4, 2.1, 2.2, etc.). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release **.** *level* **.** *branch* **.** *sequence*

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, etc. This is shown in Figure 14-2.

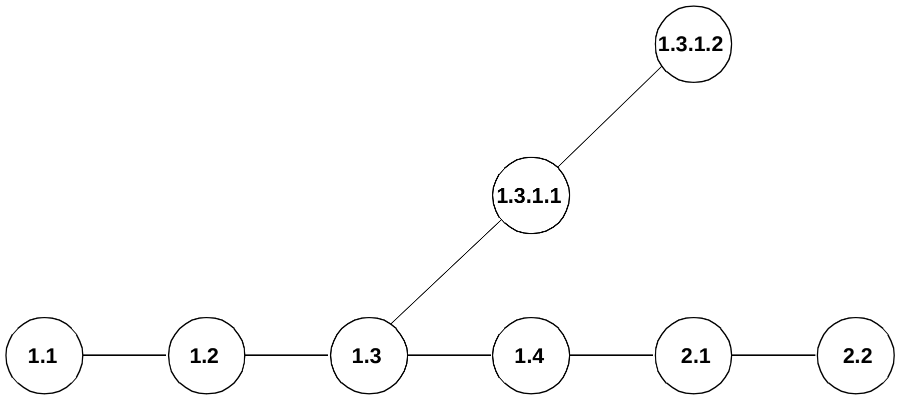


Figure 14-2. Tree Structure with Branch Deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 14-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, etc.), but the secondary branch number remains the same.

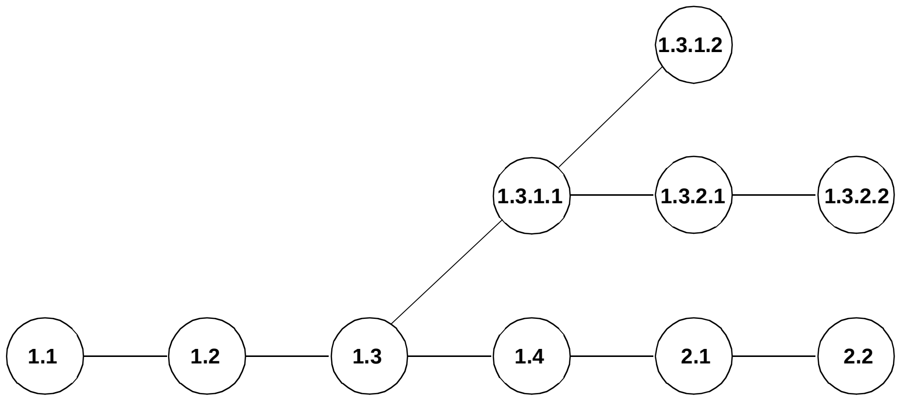


Figure 14-3. Extended Branching Concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments, key letters and file names. Key letters are options that begin with a hyphen (`-`) followed by a lowercase letter and, in some cases, a value.

File and/or directory names specify the file(s) the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files in the named directories are silently ignored.

In general, file name arguments may not begin with a hyphen. If a lone hyphen is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines.

Key letters are processed before file names, so the placement of key letters is arbitrary — they may be interspersed with file names. File names, however, are processed left to right. Somewhat different conventions apply to **`help`**, **`what`**, **`sccsdiff`**, and **`val`**, detailed later in “SCCS Commands” on page 14-8.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but for a complete description see **`admin(1)`**.

The distinction between real user (see **`passwd(1)`**) and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are the same — the person logged into the UNIX system.

x.files and z.files

All SCCS commands that modify an SCCS file do so by first writing and modifying a copy called **`x.`***file*. This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. **`x.`***file* is created in the same directory as the SCCS file, given the same mode (see **`chmod(1)`**) and is owned by the effective user. It exists only for the duration of the execution of the command that creates it. When processing is complete, the contents of **`s.`***file* are replaced by the contents of **`x.`***file*, whereupon **`x.`***file* is destroyed.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called **`z.`***file*. **`z.`***file* contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. **`z.`***file* is created with access permission mode 444 (read-only for owner, group, and other) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, you can ignore these files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

`ERROR [`*file*`]:` *message text* `(`*code*`)`

The code in parentheses can be used as an argument to the **`help`** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments.

Here is a quick-reference overview of the commands:

`get(1)`	Retrieves versions of SCCS files.
`unget(1)`	Undoes the effect of a **`get -e`** prior to the file being **`delta`**ed.
`delta(1)`	Applies deltas (changes) to SCCS files and creates new versions.

admin(1)	Initializes SCCS files, manipulates their descriptive text, and controls delta creation rights.
prs(1)	Prints portions of an SCCS file in user-specified format.
sact(1)	Prints information about files that are currently out for editing.
help(1)	Gives explanations of error messages.
rmdel(1)	Removes a delta from an SCCS file — allows removal of deltas created by mistake.
cdc(1)	Changes the commentary associated with a delta.
what(1)	Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it — useful in finding identifying information inserted by the **get** command.
sccsdiff(1)	Shows differences between any two versions of an SCCS file.
comb(1)	Combines consecutive deltas into one to reduce the size of an SCCS file.
val(1)	Validates an SCCS file.

The get Command

The **get** command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file, called a *g-file* (for gotten), is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the **get** command is used.

The most common use of **get** is

```
get s.abc
```

which normally retrieves the latest version of **s.abc** from the SCCS file tree trunk and produces (for example) on the standard output

```
1.3
67 lines
No id keywords (cm7)
```

meaning version 1.3 of **s.abc** was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The *g-file*, namely, file **abc**, is given access permission mode 444 (read-only for owner, group, and other). This particular way of using **get** is intended to produce *g-file*s only for inspection, compilation, or copying, for example. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example,

```
get s.abc s.xyz
```

produces

```
s.abc:
1.3
67 lines
No id keywords (cm7)
s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a *g-file* for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, and so on in the *g-file* itself. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the *g-file* are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example,

```
%I%
```

is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, `%H%` and `%M%` are the date and name of the *g-file*, respectively. Thus, executing **`get`** on an SCCS file that contains the PL/I declaration

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT(‘MODNAME 2.3 07/18/85’);
```

When no ID keywords are substituted by **`get`**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **`get`** although the presence of the **`i`** flag in the SCCS file causes it to be treated as an error. For a complete list of the keywords provided, see **`get(1)`**.

Retrieval of Different Versions

The version of an SCCS file that **`get`** retrieves by default is the most recently created delta of the highest numbered trunk release. However, any other version can be retrieved with **`get -r`** by specifying the version's SID. Thus,

```
get -r1.3 s.abc
```

retrieves version 1.3 of **`s.abc`** and produces (for example) on the standard output

```
1.3
64 lines
```

A branch delta may be retrieved similarly,

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output

```
1.5.2.3
234 lines
```

When a SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file **s.abc** and release 7 is the highest-numbered release below 9. Executing

```
get -r9 s.abc
```

would produce

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file **s.abc** below release 9. Similarly, omitting the sequence number, as in

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch. This might result in the following output:

```
4.3.2.8
89 lines
```

(If the given branch does not exist, an error message results.)

get -t will retrieve the latest (top) version of a particular release when no **-r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

would produce

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5
46 lines
```

To Update Source

`get -e` indicates an intent to make a delta. First, **`get`** checks the following:

- The user list to determine if the login name or group ID of the person executing **`get`** is present. The login name or group ID must be present for the user to be allowed to make deltas. (See “The admin Command” on page 14-19 for a discussion of making user lists.)
- The release number (R) of the version being retrieved to determine if the release being accessed is a protected release. That is, the release number must satisfy the relation

  floor `is less than or equal to R,`
  `which is less than or equal to` *ceiling*

  floor and *ceiling* are flags in the SCCS file representing start and end of the range of valid releases.
- R is not locked against editing. The lock is a flag in the SCCS file.
- Whether multiple concurrent edits are allowed for the SCCS file by the **`j`** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **`get -e`** causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) that is owned by the real user. If a writable *g-file* already exists, **`get`** terminates with an error.

Any ID keywords appearing in the *g-file* are not replaced by **`get -e`** because the generated *g-file* is subsequently used to create another delta.

In addition, **`get -e`** causes the creation (or updating) of the **`p.`***file* that is used to pass information to the **`delta`** command.

The following

```
get -e s.abc
```

produces (for example) on the standard output

```
1.3
new delta 1.4
67 lines
```

Undoing a get -e

There may be times when a file is retrieved accidentally for editing; there is really no editing that needs to be done at this time. In such cases, the **`unget`** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **`get -r`** and/or **`-t`** are used together with **`-e`**, the version retrieved for editing is the one specified with **`-r`** and/or **`-t`**.

`get -i` and **`-x`** are used to specify a list of deltas to be included and excluded, respectively (see **`get(1)`** for the syntax of such a list). Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **`get`** checks for possible interference with other deltas. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. A warning shows the range of lines within the retrieved *g-file* where the problem may exist. The user should examine the *g-file* to determine what the problem is and take appropriate corrective steps (edit the file if necessary).

CAUTION

`get -i` and **`get -x`** should be used with extreme care.

`get -k` is used either to regenerate a *g-file* that may have been accidentally removed or ruined after **`get -e`**, or simply to generate a *g-file* in which the replacement of ID keywords has been suppressed. A *g-file* generated by **`get -k`** is identical to one produced by **`get -e`**, but no processing related to **`p.`***file* takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **`get -e`** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The **`p.`***file* created by **`get -e`** is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. It contains the following information for each delta that is still in progress:

- The SID of the retrieved version
- The SID given to the new delta when it is created
- The login name of the real user executing **`get`**

The first execution of **`get -e`** causes the creation of **`p.`***file* for the corresponding SCCS file. Subsequent executions only update **`p.`***file* with a line containing the above information. Before updating, however, **`get`** checks to assure that no entry already in **`p.`***file* specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of **`get`** must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the *g-file*, which is an SCCS error condition. In practice, this problem does not arise because each user normally has a different working directory. See “Protection” on page 14-26 for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 14-1 shows the possible SID components a user can specify with **`get`** (left-most column), the version that will then be retrieved by **`get`**, and the resulting SID for the delta, which **`delta`** will create (right-most column). In the table

- R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.
- The **`-b`** key letter is effective only if the **`b`** flag (see **`admin(1)`**) is present in the file. An entry of `-` means irrelevant.
- The first two entries in the left-most column apply only if the **`d`** (default SID) flag is not present. If the **`d`** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this figure applies.
- R.1 (the third entry in the right-most column) is used to force the creation of the first delta in a new release.
- hR (the seventh entry in the fourth column) is the highest existing release that is lower than the specified, nonexistent release R.

Table 14-1. Determination of New SID

SID Specified in **`get`**	**`-b`** Key-Letter Used	Other Conditions	SID Retrieved by **`get`**	SID of Delta To be Created by **`delta`**
none	no	R defaults to mR	mR.mL	mR.(mL+1)
none	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1

Table 14-1. Determination of New SID (Cont.)

SID Specified in **get**	**-b** Key-Letter Used	Other Conditions	SID Retrieved by **get**	SID of Delta To be Created by **delta**
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	-	R< mR and R does not exist	hR.mL	hR.mL.(mB+1).1
R	-	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk successor	R.L	R.(L+1)
R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	-	Trunk successor in release R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	-	Branch successor	R.L.B.S	R.L.(mB+1).1

Concurrent Edits of Same SID

Under normal conditions, more than one **get -e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get -e** is executed on the same SID.

Multiple concurrent edits are allowed if the `j` flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening **`delta`**. In this case, a **`delta`** after the first **`get`** will produce delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **`delta`** after the second **`get`** will produce delta 1.1.1.1.

Key letters that Affect Output

`get -p` causes the retrieved text to be written to the standard output rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the standard error. **`get -p`** is used, for example, to create a *g-file* with an arbitrary name, as in

`get -p s.abc >` *arbitrary file name*

`get -s` suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the standard error. **`get -s`** is used to prevent non-diagnostic messages from appearing on the user's terminal and is often used with **`-p`** to pipe the output, as in

```
get -p -s s.abc | pg
```

`get -g` prints the SID on standard output and there is no retrieval of the SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file

```
get -g -r4.3 s.abc
```

outputs the SID 4.3 if it exists in the SCCS file **`s.abc`** or an error message if it does not. Another use of **`get -g`** is in regenerating a **`p.`***file* that may have been accidentally destroyed, as in

```
get -e -g s.abc
```

`get -l` causes SCCS to create **`l.`***file* in the current directory with mode 444 (read-only for owner, group, and other) and owned by the real user. The **`l.`***file* contains a table (whose format is described on **`get(1)`**). showing the deltas used in constructing a particular version of the SCCS file. For example

```
get -r2.3 -l s.abc
```

generates an **`l.`***file* showing the deltas applied to retrieve version 2.3 of **`s.abc`**. Specifying **`p`** with **`-l`**, as in

```
get -lp -r2.3 s.abc
```

causes the output to be written to the standard output rather than to **`l.`***file*. **`get -g`** can be used with **`-l`** to suppress the retrieval of the text.

`get -m` identifies the changes applied to an SCCS file. Each line of the *g-file* is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

`get -n` causes each line of a *g-file* to be preceded by the value of the `%M%` ID keyword and a tab character. This is most often used in a pipeline with **`grep(1).`** For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

`get -p -n -s` *directory* **`|`** **`grep`** *pattern*

If both **`-m`** and **`-n`** are specified, each line of the *g-file* is preceded by the value of the `%M%` ID keyword and a tab (this is the effect of **`-n`**) and is followed by the line in the format produced by **`-m`**.

Because use of **`-m`** and/or **`-n`** causes the contents of the *g-file* to be modified, such a *g-file* must not be used for creating a delta. Therefore, neither **`-m`** nor **`-n`** may be specified together with **`get -e`**. See the **`get(1)`** page.

The delta Command

The **`delta`** command is used to incorporate changes made to a *g-file* into the corresponding SCCS file — that is, to create a delta and, therefore, a new version of the file.

The **`delta`** command requires the existence of **`p.`***file* (created by **`get -e`**). It examines **`p.`***file* to verify the presence of an entry containing the user's login name. If none is found, an error message results.

The **`delta`** command performs the same permission checks that **`get -e`** performs. If all checks are successful, **`delta`** determines what has been changed in the *g-file* by comparing it with its own temporary copy of the *g-file* as it was before editing. This temporary copy is called **`d.`***file* and is obtained by performing an internal **`get`** on the SID specified in the **`p.`***file* entry.

The required **`p.`***file* entry is the one containing the login name of the user executing **`delta`**, because the user who retrieved the *g-file* must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **`get -e`** more than once on the same SCCS file. Then, **`delta -r`** must be used to specify the SID that uniquely identifies the **`p.`***file* entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **`delta`** is

```
delta s.abc
```

which prompts

```
comments?
```

to which the user replies with a description of why the delta is being made, ending the reply with a new-line character. The user's response may be up to 512 characters long with new-lines (not intended to terminate the response) escaped by backslashes (\).

If the SCCS file has a **`v`** flag, **`delta`** first prompts with

```
MRs?
```

(Modification Requests) on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a new-line character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, and so on, collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

`delta -y` and/or **`-m`** can be used to enter comments and MR numbers on the command line rather than through the standard input, as in

> **`delta -y`***"descriptive comment"* **`-m`***"mrnum1 mrnum2"* **`s.abc`**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two key letters are useful when **`delta`** is executed from within a shell procedure. Note that **`delta -m`** is allowed only if the SCCS file has a **`v`** flag.

No matter how comments and MR numbers are entered with **`delta`**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **`delta`**.

If **`delta`** is used with more than one file argument and the first file named has a **`v`** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **`delta`** processing is complete, the standard output displays the SID of the new delta (from **`p.`***file*) and the number of lines inserted, deleted, and left unchanged. For example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a *g-file*, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the *g-file*. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited *g-file*.

If you are in the process of making a delta and the **`delta`** command finds no ID keywords in the edited *g-file*, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a *g-file* that was created by a **`get`** without **`-e`** (ID keywords are replaced by **`get`** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the *g-file*. Or, it is possible that the file had no ID keywords. In any case, the delta will be created unless there is an **`i`** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding **`p.`***file* entry is removed from **`p.`***file*. All updates to **`p.`***file* are made to a temporary copy, **`q.`***file*, whose use is similar to that of **x.***file* described in “SCCS Command Conventions” on page 14-7. If there is only one entry in **p.***file*, then **p.***file* itself is removed.

In addition, **delta** removes the edited *g-file* unless **-n** is specified. For example

```
delta -n s.abc
```

will keep the *g-file* after processing.

delta -s suppresses all output normally directed to the standard output, other than `comments?` and MRs?. Thus, use of **-s** with **-y** (and/or **-m**) causes **delta** neither to read from the standard input nor to write to the standard output.

The differences between the *g-file* and the **d.***file* constitute the delta and may be printed on the standard output by using **delta -p**. The format of this output is similar to that produced by **diff**.

The admin Command

The **admin** command is used to administer SCCS files — that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of key letters with **admin** or are assigned default values if no key letters are supplied. The same key letters are used to change the parameters of existing SCCS files.

Two key letters are used in detecting and correcting corrupted SCCS files (see “Auditing” on page 14-28).

Newly created SCCS files are given access permission mode 444 (read-only for owner, group and other) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin(1)** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command

```
admin -ifirst s.abc
```

in which the value `first` with **-i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **-i** means **admin** is to read the standard input for the text of the initial delta.

The command

```
admin -i s.abc < first
```

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **`admin`** as a warning. However, if the command also sets the **`i`** flag (not to be confused with the **`-i`** key letter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **`admin -i`**.

`admin -r` is used to specify a release number for the first delta. Thus:

```
    admin -ifirst -r3 s.abc
```

means the first delta should be named 3.1 rather than the normal 1.1. Because **`-r`** has meaning only when creating the first delta, its use is permitted only with **`-i`**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**`admin -y`**) and/or MR numbers (**`-m`**) can be entered in exactly the same way as with **`delta`**.

If **`-y`** is omitted, a comment line of the form

date and time created YY/MM/DD HH:MM:SS by logname

is automatically generated.

If it is desired to supply MR numbers (**`admin -m`**), the **`v`** flag must be set with **`-f`**. The **`v`** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary in the SCCS file (see **`sccsfile(4)`**). An example would be

```
    admin -ifirst -mmrnum1 -fv s.abc
```

Note that **`-y`** and **`-m`** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **`admin -t`**.

When an SCCS file is first being created and **`-t`** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
    admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file **`desc`**.

When processing an existing SCCS file, **`-t`** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
    admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of **`desc`**. Omission of the file name after the **`-t`** key letter as in

```
    admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **`admin -f`**, or deleted by **`admin -d`**.

SCCS file flags are used to direct certain actions of the various commands. (See the **`admin(1)`** page for a description of all the flags.) For example, the **`i`** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **`d`** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **`get`** command.

`admin -f` is used to set flags and, if desired, their values. For example

```
admin -ifirst -fi -fmmodname s.abc
```

sets the **`i`** and **`m`** (module name) flags. The value *modname* specified for the **`m`** flag is the value that the **`get`** command will use to replace the `%M%` ID keyword. (In the absence of the **`m`** flag, the name of the *g-file* is used as the replacement for the `%M%` ID keyword.) Several **`-f`** key letters may be supplied on a single **`admin`**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

`admin -d` is used to delete a flag from an existing SCCS file. As an example, the command

```
admin -dm s.abc
```

removes the **`m`** flag from the SCCS file. Several **`-d`** key letters may be used with one **`admin`** and may be intermixed with **`-f`**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **`admin -a`** is used. For example,

```
admin -axyz -awql -a1234 s.abc
```

adds the login names `xyz` and `wql` and the group ID `1234` to the list. **`admin -a`** may be used whether creating a new SCCS file or processing an existing one.

`admin -e` (erase) is used to remove login names or group IDs from the list.

The prs Command

The **`prs`** command is used to print all or part of an SCCS file on the standard output. If **`prs -d`** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data keywords (not to be confused with **`get`** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example,

```
:I:
```

is defined as the data keyword replaced by the SID of a specified delta. Similarly, `:F:` is the data keyword for the SCCS file name currently being processed, and `:C:` is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list, see the **`prs(1)`** page.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example,

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **`prs -r`**. For example,

```
prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **`-r`** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **`-l`** or **`-e.`** The use of **`prs -e`** substitutes data keywords for the SID designated with **`-r`** and all deltas created earlier, while **`prs -l`** substitutes data keywords for the SID designated with **`-r`** and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **`-e`** and **`-l`**.

The sact Command

`sact` is a special form of the **`prs`** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **`get -e`** command, and the date and time the **`get -e`** was executed. It is a useful command for an administrator.

The help Command

The **`help`** command prints information about messages that may appear on the user's terminal. Arguments to **`help`** are the code numbers that appear in parentheses at the end of SCCS messages. (If no argument is given, **`help`** prompts for one.) Explanatory information is printed on the standard output. If no information is found, an error message is printed. When more than one argument is used, each is processed independently, and an error resulting from one will not stop the processing of the others. For more information, see the **`help(1)`** page.

The rmdel Command

The **`rmdel`** command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 14-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **`-r`** key letter is mandatory with **`rmdel`**. It is used to specify the complete SID of the delta to be removed. Thus

```
rmdel -r2.3 s.abc
```

specifies the removal of trunk delta 2.3.

Before removing the delta, **`rmdel`** checks that the release number (R) of the given SID satisfies the relation

```
floor is less than or equal to R,
which is less than or equal to ceiling
```

floor and *ceiling* are flags in the SCCS file representing start and end of the range of valid releases.

The **`rmdel`** command also checks the SID to make sure it is not for a version on which a **`get`** for editing has been executed and whose associated **`delta`** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing.

That is, if the **l** flag is set (see **admin(1)**), the release must not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from `D` (delta) to `R` (removed).

The cdc Command

The **cdc** command is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (for example, **-r** and full SID are necessary), although the delta need not be a leaf delta. For example,

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old MR numbers with the `!` symbol. Thus

```
cdc -r1.4 s.abc
MRs?  mrnum3 !mrnum1       (The MRs? prompt appears only
                           if the v flag has been set.)
comments? deleted wrong MR no.and inserted correct MR no.
```

inserts `mrnum3` and deletes `mrnum1` for delta 1.4.

The what Command

The **what** command is used to find identifying information in any UNIX system file whose name is given as an argument. No key letters are accepted. The **what** command searches the given file(s) for all occurrences of the string `@(#)`, which is the replacement for the `%Z%` ID keyword (see the **get(1)** page). It prints on the standard output whatever follows the string until the first double quote (“), greater than symbol (>), backslash (\), new-line, null, or non-printing character.

For example, if an SCCS file called **s.prog.c** (a C language source file) contains the following line

```
char id[]= "%W%";
```

and the command

```
get -r3.4 s.prog.c
```

is used, the resulting *g-file* is compiled to produce **prog.o** and **a.out**. Then, the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:  3.4
prog.o:
  prog.c:  3.4
a.out:
  prog.c:  3.4
```

The string searched for by **what** need not be inserted with an ID keyword of **get**; it may be inserted in any convenient manner.

The sccsdiff Command

The **sccsdiff** command determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff -r** in the same way as with **get -r**. SID numbers must be specified as the first two arguments. The SCCS file or files to be processed are named last. Directory names and a lone hyphen are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

The differences are printed the same way as by **diff**.

The comb Command

The **comb** command lets the user reduce the size of an SCCS file. It generates a shell procedure on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any key letters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 14-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the key letters used with this command are:

-s	This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as a preliminary check.
-p	This option is used to specify the oldest delta the user wants preserved.

-c This option is used to specify a list (see the **get(1)** page for its syntax) of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val Command

The **val** command is used to determine whether a file is an SCCS file meeting the characteristics specified by certain key letters. It checks for the existence of a particular delta when the SID for that delta is specified with **-r**.

The string following **-y** or **-m** is used to check the value set by the **t** or **m** flag, respectively. See **admin(1)** for descriptions of these flags.

The **val** command treats the special argument hyphen differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file (control-d) is entered. This permits one **val** command with different values for key letters and file arguments. For example,

```
val -
-yc -mabc s.abc
-mxyz -ypl1 s.xyz
control_d
```

first checks if file **s.abc** has a value c for its type flag and value abc for the module name flag. Once this is done, **val** processes the remaining file, in this case **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error (see **val(1)** for a description of errors and codes). In addition, an appropriate diagnostic is printed unless suppressed by **-s**. A return code of 0 means all files met the characteristics specified.

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the UNIX system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files — that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **`admin`** command are given access permission mode 444 (read-only for owner, group, and other). This mode should remain unchanged because it (generally) prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings — subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating and modifying a copy of the file. When processing is done, the contents of the old file are automatically replaced by the contents of the copy, whereupon the copy is destroyed. If the old file had additional links, this would break them. Then, rather than process such files, SCCS commands would produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (on large development projects, for example), one user — that is, one user ID — must be chosen as the owner of the SCCS files. This person will administer the files (use the **`admin`** command) and will be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **`get`**, **`delta`**, and, if desired, **`rmdel`** and **`cdc`** commands.

The interface program must be owned by the SCCS administrator and must have the set-user-ID-on-execution bit on (see **`chmod(1)`**). This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **`delta`** and, possibly, **`rmdel`** and **`cdc`**.

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	a line containing the logical sum of all the characters of the file (not including the checksum line itself)
Delta Table	information about each delta, such as type, SID, date and time of creation, and commentary
User Names	list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas
Flags	indicators that control certain actions of SCCS commands

Descriptive Text	usually a summary of the contents and purpose of the file
Body	the text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **`sccsfile(4).`** The checksum line is discussed in “Auditing” on page 14-28.

Because SCCS files are ASCII files they can be processed by non-SCCS commands like **`ed`**, **`grep`**, and **`cat`**. This is convenient when an SCCS file must be modified manually (a delta's time and date were recorded incorrectly, for example, because the system clock was set incorrectly), or when a user wants simply to look at the file.

CAUTION

Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted because it was last accessed (possibly by having lost one or more blocks or by having been modified with **`ed`**). No SCCS command will process a corrupted SCCS file except the **`admin -h`** or **`-z`**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **`admin -h`** and specify all SCCS files:

`admin -h s.`_file1_ **`s.`**_file2_ . . .

or

`admin -h` _directory1 directory2_ . . .

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **`ls`** command periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **`admin`** command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

Index

Spine for 1.5” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

Programmer

Compilaton Systems
Volume 1 (Tools)

0890459