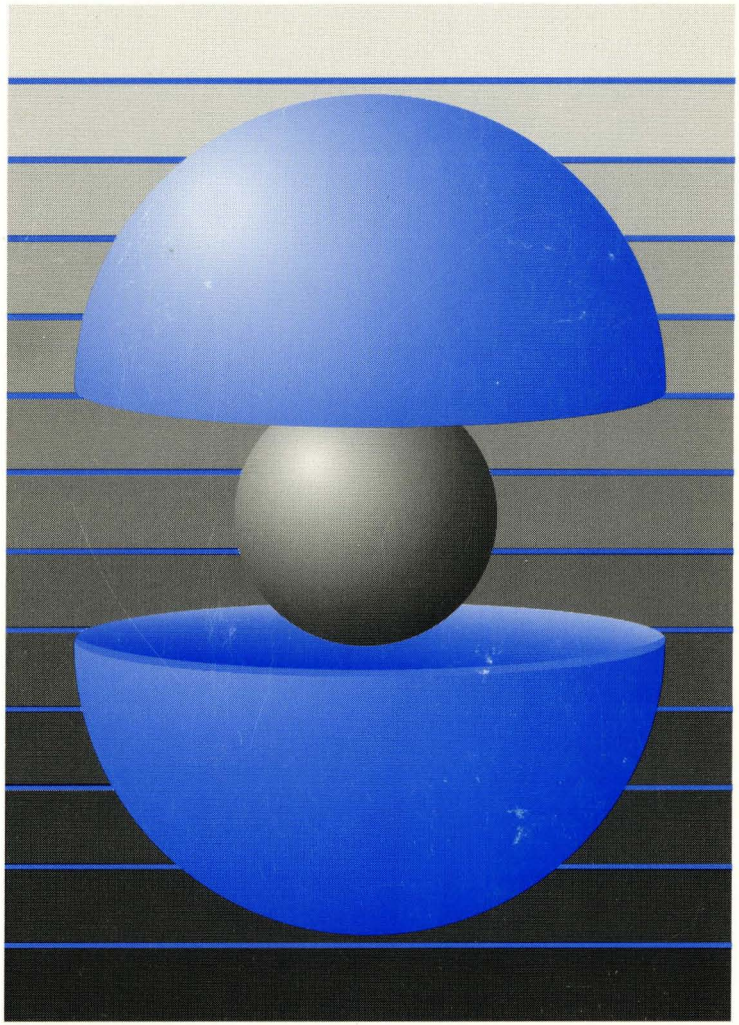
DEC OSF/1

digital

Programming Support Tools



Part Number: AA-PS32B-TE

DEC OSF/1

Programming Support Tools

Order Number: AA-PS32B-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This manual describes commands and utilities for assisting in program development and in building and installing software product kits.

digital equipment corporation
Maynard, Massachusetts

Contents

About This Manual

Audience xv

Organization xv

Related Documents xvi

Reader's Comments xvi

Conventions xvii

1 Finding Information with Regular Expressions and the grep Commands

1.1 Forming Regular Expressions 1–1

1.1.1 Matching Multiple Occurrences of a Regular Expression 1–3

1.1.2 Matching Only Selected Characters 1–5

1.1.3 Specifying Multiple Regular Expressions 1–5

1.1.4 Special Collating Considerations in Regular Expressions 1–5

1.2 Using the grep Commands 1–6

2 Matching Patterns and Processing Information with awk

2.1 Running the awk Program 2–2

2.2 Printing in awk 2–5

2.3 Using Variables in awk 2–6

2.4 Performing Actions Before or After Processing the Input 2–8
2.5 Using Regular Expressions as Patterns 2–8
2.6 Using Relational Expressions and Combined Expressions as Patterns . 2–9
2.7 Using Pattern Ranges 2–10
2.8 Actions in awk 2–11
2.9 Concatenating Strings 2–11
2.10 Using Variables in an Action 2–12
2.10.1 Simple Variables 2–12
2.10.2 Field Variables 2–13
2.10.3 Array Variables 2–13
2.11 Using Operators in an Action 2–14
2.12 Using Functions Within an Action 2–15
2.13 Using Control Structures in awk 2–18
2.14 Redirection and Pipes 2–20

3 Editing Files with the sed Editor

3.1 Running the sed Editor 3–1
3.2 Selecting Lines for Editing 3–4
3.3 Summary of sed Commands 3–6
3.4 String Replacement 3–12

4 Creating Input Language Analyzers and Parsers

4.1 How the Lexical Analyzer Works 4–1
4.2 Writing a Lexical Analyzer Program with lex 4–2
4.3 The lex Specification File 4–3
4.3.1 Defining Substitution Strings 4–4
4.3.2 Rules 4–4
4.3.2.1 Regular Expressions 4–5

4.3.2.1.1 Including Blanks in an Expression 4–7
4.3.2.1.2 Other Special Characters 4–7
4.3.2.2 Matching Rules 4–8
4.3.2.2.1 Using Wildcard Characters to Match a String . 4–8
4.3.2.2.2 Finding Strings Within Strings 4–9
4.3.2.3 Actions 4–10
4.3.2.3.1 Null Action 4–10
4.3.2.3.2 Using the Same Action for Multiple Expressions 4–10
4.3.2.3.3 Printing a Matched String 4–10
4.3.2.3.4 Finding the Length of a Matched String 4–11
4.3.2.3.5 Getting More Input 4–11
4.3.2.3.6 Returning Characters to the Input 4–12
4.3.3 Using or Overriding Standard Input/Output Routines 4–12
4.3.4 End-of-File Processing 4–14
4.3.5 Passing Code to the Generated Program 4–14
4.3.6 Start Conditions 4–15
4.4 Generating a Lexical Analyzer 4–15
4.5 Using lex with yacc 4–16
4.6 Creating a Parser with yacc 4–18
4.6.1 The main and yyerror Functions 4–19
4.6.2 The yylex Function 4–20
4.7 The Grammar File 4–21
4.7.1 Declarations 4–21
4.7.1.1 Defining Global Variables 4–22
4.7.1.2 Start Symbols 4–23
4.7.1.3 Token Numbers 4–23
4.7.2 Grammar Rules 4–24
4.7.2.1 The Null String 4–24
4.7.2.2 End-of-Input Marker 4–24
4.7.2.3 Actions in yacc Parsers 4–25
4.7.3 Programs 4–26

4.7.4 Guidelines for Using Grammar Files 4–26
4.7.4.1 Using Comments 4–27
4.7.4.2 Using Literal Strings 4–27
4.7.4.3 Guidelines for Formatting the Grammar File 4–27
4.7.4.4 Using Recursion in a Grammar File 4–28
4.7.4.5 Errors in the Grammar File 4–28
4.7.5 Error Handling by the Parser 4–28
4.7.5.1 Providing for Error Correcting 4–29
4.7.5.2 Clearing the Look-Ahead Token 4–30
4.8 Parser Operation 4–30
4.8.1 The shift Action 4–31
4.8.2 The reduce Action 4–31
4.8.3 Ambiguous Rules and Parser Conflicts 4–32
4.9 Turning on Debug Mode 4–34
4.10 Creating a Simple Calculator Program 4–35
4.10.1 The Parser Source Code 4–36
4.10.2 The Lexical Analyzer Source Code 4–39

5 Using m4 Macros in Your Programs

5.1 Using Macros 5–1
5.2 Defining Macros 5–2
5.2.1 Using the Quote Characters 5–3
5.2.2 Macro Arguments 5–5
5.3 Using Other m4 Macros 5–6
5.3.1 Changing the Comment Characters 5–8
5.3.2 Changing the Quote Characters 5–9
5.3.3 Removing a Macro Definition 5–9
5.3.4 Checking for a Defined Macro 5–9
5.3.5 Using Integer Arithmetic 5–9
5.3.6 Manipulating Files 5–10
5.3.7 Redirecting Output 5–11

5.3.8 Using System Programs in a Program 5–11
5.3.9 Using Unique File Names 5–11
5.3.10 Using Conditional Expressions 5–12
5.3.11 Manipulating Strings 5–12
5.3.12 Printing 5–13

6 Revision Control: Managing Source Files with RCS or SCCS

6.1 Version Control Concepts 6–3
6.2 Managing Multiple Versions of Files 6–6
6.3 Creating a Version Control Library 6–8
6.4 Using RCS 6–8
6.4.1 Placing New Files in an RCS Library 6–10
6.4.2 Recording File-Indentification Information with RCS 6–11
6.4.3 Getting Files from an RCS Library 6–12
6.4.4 Checking Edited Files Back into an RCS Library 6–13
6.4.5 Working with Multiple Versions of Files 6–13
6.4.6 Displaying Differences in RCS Files 6–15
6.4.7 Reporting Revision Histories of RCS Files 6–15
6.4.8 Configuration Control Concepts 6–17
6.5 Using SCCS 6–18
6.5.1 Placing New Files in an SCCS Library 6–20
6.5.2 Recording File-Identification Information with SCCS 6–21
6.5.3 Getting Files from an SCCS Library 6–22
6.5.3.1 Getting Files for Purposes Other Than Editing 6–23
6.5.3.2 Getting Files for Editing 6–23
6.5.3.3 Managing Multiple Files and New Releases 6–24
6.5.4 Checking Edited Files Back into an SCCS Library 6–24
6.5.5 Working with Multiple Versions of Files 6–25
6.5.6 Displaying Differences in SCCS Files 6–26
6.5.7 Reporting Revision Histories of SCCS Files 6–27
6.5.8 Performing Administrative Functions 6–27
6.5.9 Using SCCS Options 6–30
6.5.10 Summary of Individual SCCS Commands 6–31

6.6 Functional Comparison of RCS and SCCS Commands 6–33

7 Building Programs with the make Utility

7.1 Operation of the make Utility 7–2
7.2 Description Files 7–3
7.2.1 Format of a Description File Entry 7–4
7.2.2 Using Commands in a Description File 7–5
7.2.3 Preventing the make Utility from Echoing Commands 7–6
7.2.4 Preventing the make Utility from Stopping on Errors 7–6
7.2.5 Defining Default Conditions 7–7
7.2.6 Preventing make from Deleting Files 7–7
7.2.7 Simple Description File 7–7
7.2.8 Making the Description File Simpler 7–8
7.2.9 Defining Macros 7–8
7.2.10 Using Macros in a Description File 7–9
7.2.10.1 Macro Substitution 7–9
7.2.10.2 Conditional Macros 7–12
7.2.11 Calling the make Utility from a Description File 7–13
7.2.12 Internal Macros 7–13
7.2.12.1 Internal Target File Name Macro 7–14
7.2.12.2 Internal Label Name Macro 7–14
7.2.12.3 Internal Younger Files Macro 7–15
7.2.12.4 Internal First Out-of-Date File Macro 7–15
7.2.12.5 Internal Current File Name Prefix Macro 7–15
7.2.13 How make Uses Environment Variables 7–15
7.2.14 Internal Rules 7–16
7.2.14.1 Single Suffix Rules 7–18
7.2.14.2 Overriding Built-In make Macros 7–19
7.2.15 Including Other Files 7–20
7.2.16 Testing Description Files 7–20
7.2.17 Description File 7–21

8 Creating and Managing Software Product Kits

8.1 The setld Command and Its Functions 8–2

8.2 Files Used by the setld Utility 8–3

8.3 Descriptions of setld Functions 8–4

8.3.1 Loading Software 8–5
8.3.2 Configuring a Subset 8–6
8.3.3 Verifying a Subset 8–6
8.3.4 Removing Software 8–7

8.4 Using the File System Effectively 8–7

8.4.1 Using Standard Directories 8–7
8.4.2 Placing Layered Product Files 8–15

8.5 Creating Kits for the setld Utility 8–17

8.5.1 Creating a Source Hierarchy 8–19
8.5.2 Creating the Kit Building Control Files 8–21

8.5.2.1 Creating the Master Inventory 8–21
8.5.2.2 Creating the Key File 8–24

8.5.3 Creating Subset Control Programs 8–27

8.5.3.1 Invoking a Subset Control Program 8–28
8.5.3.2 Managing Subset Dependencies 8–30

8.5.3.2.1 Dependency Expressions 8–31
8.5.3.2.2 Using the STL_DepInit Routine 8–32
8.5.3.2.3 Using the STL_DepEval Routine 8–32
8.5.3.2.4 Using the STL_ArchAssert Routine 8–32
8.5.3.2.5 Using the STL_LockInit Routine 8–32
8.5.3.2.6 Using the STL_DepLock Routine 8–32
8.5.3.2.7 Using the STL_DepUnLock Routine 8–33

8.5.3.3 Using Control File Flag Bits 8–33
8.5.3.4 An Example Subset Control Program 8–33
8.5.3.5 Creating Symbolic Links for Layered Products 8–36

8.5.3.5.1 Creating Standard (Forward) Symbolic Links . 8–37
8.5.3.5.2 Creating Backward Links 8–39
8.5.3.5.3 Using the STL_LinkInit Routine 8–39
8.5.3.5.4 Using the STL_LinkBack Routine 8–39

8.5.4 Building Your Kit 8–40
8.5.4.1 The Compression Flag File 8–42
8.5.4.2 The Image Data File 8–42
8.5.4.3 The Subset Control Files 8–42
8.5.4.4 The Subset Inventory Files 8–43
8.5.5 Transferring Your Kit to Distribution Media 8–45
8.5.5.1 Building a Kit on Magnetic Tape 8–45
8.5.5.2 Building a Kit on a Disk 8–45
8.5.6 Installing and Distributing Kits on a Network 8–46

Glossary

Index

Examples

4-1: Parser Source Code for a Calculator 4–36
4-2: Lexical Analyzer Source Code for a Calculator 4–39
7-1: A Simple Description File 7–8
7-2: Default Rules File 7–18
7-3: The makefile for the make Utility 7–21
8-1: Master Inventory File 8–22
8-2: Key File 8–24
8-3: Sample Subset Control Program 8–34
8-4: Sample Link Control Program 8–37
8-5: Example of Backward link Creation 8–40
8-6: Sample Subset Inventory File 8–43

Figures

2-1: Sequence of awk Processing 2–5
3-1: Sequence of sed Processing 3–4
4-1: Simple Finite State Model 4–2
4-2: Producing an Input Parser with lex and yacc 4–17
6-1: Contents of a Version Control File 6–4
6-2: A Typical RCS Library 6–5
6-3: A Typical SCCS Library 6–6
6-4: A Version Control File's Tree of Deltas 6–7
8-1: Base System Directory Hierarchy 8–8
8-2: X Directory Hierarchy 8–12
8-3: How Layered Products Are Installed 8–16
8-4: The Kit Building Process 8–18
8-5: Directory Hierarchies for a Kit 8–18
8-6: Directory Hierarchy for the DCB Kit 8–20

Tables

1-1: Rules for Regular Expressions 1–1
1-2: Versions of the grep Command 1–6
1-3: Options for the grep Utilities 1–7
2-1: Options for the awk and gawk Commands 2–2
2-2: Built-In Variables in awk 2–6
2-3: Operators for awk Actions 2–14
2-4: Built-In awk Functions 2–16
2-5: Control Structures in awk 2–19
3-1: Options for the sed Command 3–2
3-2: Regular Expressions Recognized by sed 3–5

3-3: Text Editing and Movement Commands 3–7
3-4: Buffer Manipulation Commands 3–10
3-5: Flow-of-Control Commands 3–10
4-1: Regular Expression Operators for lex 4–5
4-2: Options for the lex Command 4–16
4-3: Processing-Condition Definition Keywords in yacc 4–22
5-1: Built-In m4 Macros 5–6
6-1: Features of RCS and SCCS 6–2
6-2: Summary of RCS Command Functions 6–8
6-3: RCS ID Keywords 6–11
6-4: Summary of sccs Command Functions 6–18
6-5: SCCS ID Keywords 6–21
6-6: SCCS admin Command Options 6–28
6-7: Flags for the admin Command 6–29
6-8: SCCS Command Options 6–31
6-9: Individual SCCS Commands 6–31
6-10: Functional Comparison: RCS and SCCS Commands 6–33
7-1: Internal make Macros 7–13
8-1: Options for the setld Command 8–2
8-2: Contents and Purposes of Base System Directories 8–9
8-3: Contents and Purposes of X Directories 8–13
8-4: Fields in Master Inventory Records 8–22
8-5: Key File Attributes Section 8–25
8-6: Key File Subset Descriptor Fields 8–26
8-7: Subset Control Program Invocation and Actions 8–28
8-8: Dependency Management Routines 8–30
8-9: Routines Assisting with Backward Link Creation 8–39
8-10: Control Files in the instctrl Directory 8–41

8-11: Image Data File Fields 8–42
8-12: Subset Inventory Field Descriptions 8–44

About This Manual

This manual describes several commands and utilities in the DEC OSF/1 system, including facilities for text manipulation, macro and program generation, source file management, and software kit installation and creation.

Audience

The commands and utilities described in this manual are intended primarily for programmers, but some of them, particularly those described in Chapter 1, Chapter 2, Chapter 3, and Chapter 6, can be very useful for writers and other types of users as well. The manual assumes that you are a moderately experienced user of UNIX systems.

Organization

This manual comprises eight chapters, a glossary, and an index. A brief description of the contents follows:

Chapter 1	Introduces the concept of regular expressions (REs) and describes the rules for forming them, and describes the `grep` family of commands that use REs for searching text files.
Chapter 2	Describes the `awk` command and its text-processing language.
Chapter 3	Describes the `sed` stream editor, a noninteractive tool for rapidly performing complex and repetitive editing tasks.
Chapter 4	Describes the `lex` and `yacc` programs for generating lexical analyzers and parsers for processing input to a program.
Chapter 5	Describes the `m4` macro preprocessor and explains how to create macros that can be used in programs or in other files such as documentation source.
Chapter 6	Describes how to manage libraries of source files by using the Source Code Control System (SCCS) or the Revision Control System (RCS).
Chapter 7	Describes how to use the `make` utility to build and maintain complex programs and applications.
Chapter 8	Describes how the `setld` software installation and management utility operates and how to create software product kits for installation with `setld`, and explains the contents of files used by `setld`.

Related Documents

This manual is an adjunct to the DEC OSF/1 *Programmer's Guide*; neither manual requires that you have the other in order to use its contents.

The printed version of the DEC OSF/1 documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General Users	G	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the books in the DEC OSF/1 documentation set.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail:
  `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed DEC OSF/1 manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

Conventions

The following typographical conventions are used in this manual:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
`#`	A number sign represents the superuser prompt.
`% cat`	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
`[ | ]` `{ | }`	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.
Return	In an example, a key name enclosed in a box indicates that you press that key.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

Finding Information with Regular Expressions and the grep Commands 1

This chapter describes regular expressions (REs) and how to use them. REs are most commonly used in the context of pattern matching with the `grep` family of pattern-matching commands, but they are also used with virtually all text-processing or filtering utilities and commands. A more thorough discussion of the `grep` commands (`grep`, `egrep`, and `fgrep`) follows the exposition of REs.

1.1 Forming Regular Expressions

An RE specifies a set of strings to be matched. It contains text characters and **operator** characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features.

The letters of the alphabet and digits are always text characters. For example, the RE `integer` matches the string "integer", and the expression `a57D` matches the string "a57D".

Text characters and operator characters together make up the set of simple REs. You can concatenate any number or combination of simple REs to create a compound RE that will match any sequence of characters that corresponds to the concatenated simple REs. Table 1-1 describes the rules for creating REs. The following sections further explain some of the REs listed in the table.

Table 1-1: Rules for Regular Expressions

Expression	Name	Description
letters, numbers, most punctuation	Text character	Matches itself.
	Period (dot)	Matches any single character except the newline character.
*	Asterisk	Matches any number of occurrences of the preceding simple RE, including none.

Table 1-1: (continued)

Expression	Name	Description
?	Question mark	Matches zero or one occurrence of the preceding simple RE. (Not available to all utilities.)
+	Plus sign	Matches one or more occurrences of the preceding simple RE. (Not available to all utilities.)
\{*expr*\}	Count expression	Matches a more restricted number of instances of the preceding simple RE; for example, `ab\{3\}c` matches only `abbbc`, while `ab\{2,3\}c` matches `abbc` or `abbbc`, but not `abc` or `abbbbc`. (Not available to all utilities.)
[*chars*]	Brackets	Matches a single instance of any one of the characters within the brackets. Ranges of characters can be abbreviated by using a dash. For example, `[0–9a–z]` matches any single digit or lowercase letter.
^	Circumflex	When used at the beginning of an RE, matches the beginning of a line. When used as the first character inside brackets, excludes the bracketed characters from being matched. Otherwise, has no special properties.
$	Dollar sign	When used at the end of an RE, matches the end of a line. Otherwise, has no special properties.
char	Backslash	Escapes the next character to permit matching on explicit instances of characters that are normally RE operators.
expr expr ...	Concatenation	Matches any string that matches all of the concatenated REs in sequence.
(*expr*)	Parentheses	Encloses, or frames, an RE, allowing operators that act on the preceding simple RE to treat the entire framed RE as a simple RE. For example, `(bc)+` matches `abcd`, `abcbcd`, `abcbcbcd`, and so forth. (Not available to all utilities.)
expr|*expr ...*	Vertical bar	Separates multiple REs; matches any of the bar-separated REs. (Not available to all utilities.)

Table 1-1: (continued)

Expression	Name	Description
`[:class:]`	Class	A character class name enclosed in bracket-colon delimiters matches any of the set of characters in the named class. Members of each of the sets are determined by the current setting of the `LC_CTYPE` environment variable. The supported classes are `alpha`, `upper`, `lower`, `digit`, `alnum`, `xdigit`, `space`, `print`, `punct`, `graph`, and `cntrl`. For example, `[[:lower:]]` matches any lowercase letter in the current collating sequence.
`\(expr\)`	Hold delimiters	Matches *expr* and saves it into a numbered holding space for reuse later in a compound RE or in a replacement string.
`\n`	Repeat expression	Repeats the contents of the `nth` set of hold delimiters in the RE.

The order of precedence of operators is brackets (`[ ]`), then the asterisk (`*`), question mark (`?`), and plus sign (`+`), then concatenation, then the vertical bar (`|`) and the newline character.

1.1.1 Matching Multiple Occurrences of a Regular Expression

An asterisk (`*`) acts on the simple RE immediately preceding it, causing that RE to match any number of occurrences of a matching pattern, even none. When an asterisk follows a period, the combination indicates a match on any sequence of characters, even none. A period and an asterisk always match as much text as possible; for example:

```
% echo "A B C D" | sed 's/^.* /E/'
ED
```

The `sed` stream editor command in this example indicates that `sed` is to match the RE between the first and second slashes and replace the matching pattern with the string between the second and third slashes. The RE will match any string that starts at the beginning of the line, contains any sequence of characters, and ends in a space. Nominally, the string "A " satisfies this expression; but the *longest* matching pattern is "A B C ", so `sed` replaces "A B C " with "E" to yield "ED" as the output. See Chapter 3 for a discussion of the `sed` stream editor.

An asterisk matches any number of instances of the preceding RE. To limit the number of instances that a particular RE will match, use a plus sign (+)

or a question mark (`?`). The plus sign requires at least one instance of its matching pattern. The question mark refuses to accept more than one instance. The following chart illustrates the matching characteristics of the asterisk, plus sign, and question mark:

Regular Expression	Matching Strings		
`ab?c`	ac	abc	
`ab*c`	ac	abc	abbc, abbbc, ...
`ab+c`		abc	abbc, abbbc, ...

You can also specify much more restrictive numbers of instances of the desired RE. Placing one or two numbers between backslash-brace delimiters (`\{` and `\}`) has the following effects:

- `\{`*number*`\}`

  Matches exactly *number* instances of anything the RE matches. For example, `ab\{3\}c` matches `abbbc` but does not match either `abbc` or `abbbbc`.

- `\{`*number*`,\}`

  Matches at least *number* instances. For example, `ab\{3,\}c` matches `abbbc`, `abbbbc`, and so on, but not `ac`, `abc`, or `abbc`.

- `\{`*number1*`,`*number2*`\}`

  Matches any number of instances from *number1* to *number2*, inclusive. For example, `ab\{2,4\}c` matches `abbc`, `abbbc`, or `abbbbc` but not `abc` or `abbbbbc`.

Using the backslash-parenthesis hold delimiters `\(` and `\)`, you can save up to nine patterns on a line. Counting from left to right on the line, the first pattern saved is placed in the first holding space, the second pattern is placed in the second holding space, and so on.

The character sequence `\`*n* (where *n* is a digit from 1 to 9) matches the *n*th saved pattern. Consider the following pattern:

```
\(A\)\(B\)C\2\1
```

This pattern matches the string `ABCBA`. You can nest patterns to be saved in holding spaces. Whether the enclosed patterns are nested or in a series, *n* refers to the *n*th occurrence, counting from the left, of the delimiters. You can also use `\`*n*expressions in replacement strings for editors such as `ed` and `sed` as well as address patterns.

1.1.2 Matching Only Selected Characters

A period in an RE matches any character except the newline character. To restrict the characters to be matched, place the desired characters inside brackets (`[ ]`). Each string of bracketed characters is a single-character RE that matches any *one* of the bracketed characters. Except for the circumflex (`^`), RE operators within brackets are interpreted literally, without special meaning. The circumflex *excludes* the bracketed characters if it is the first character in the brackets; otherwise, it has no special meaning.

When you specify a range of characters with a dash (for example, `[a–z]`), the characters that fall within the range are determined by the current collating sequence defined by the current setting of the `LC_CTYPE` environment variable. See the discussion on using internationalization features in *Command and Shell User's Guide* for more information on collating sequences. The dash has no special meaning if it is the first or last character in a bracketed string or if it immediately follows a circumflex that is the first character in a bracketed string. To include a right bracket in a bracketed string, place it first or after the initial circumflex.

You can use the `–i` option to the `grep` and `egrep` utilities to perform a case insensitive match. (The `–y` option is a synonym for `–i`.) To create an RE that is not case sensitive for other utilities, or to form an RE that is only partially case insensitive, use a bracketed RE consisting of just the uppercase and lowercase versions of the character you want. For example:

```
% grep '[Jj]ones' group-list
```

1.1.3 Specifying Multiple Regular Expressions

Some utilities, such as `egrep` and `awk`, permit you to specify multiple REs simultaneously by separating the REs with a vertical bar. For example:

```
% awk '/[Bb]lack|[Ww]hite/ {print NR ":", $0}' .Xdefaults
55: sm.pointer_foreground:  black
56: sm.pointer_background:  white
```

1.1.4 Special Collating Considerations in Regular Expressions

A character range can include a multicharacter collating element enclosed within bracket-period delimiters (`[.` and `.]`). These **collating symbols** are necessary for languages that treat some strings as individual collating elements. For example, in Spanish, the strings `ch` and `ll` each are collating symbols (that is, the Spanish primary sort order is `a`, `b`, `c`, `ch`, `d`,..., `k`, `l`, `ll`, `m`, `...`). The bracket-period delimiters in the RE syntax distinguish multicharacter collating elements from a list of the individual characters that make up the element. When using Spanish collation rules, `[[.ch.]]` is

treated as an RE matching the sequence `ch`, while `[ch]` is treated as an RE matching `c` or `h`. In addition, `[a-[.ch.]]` matches `a`, `b`, `c`, and `ch`.

A collating sequence can define equivalence classes for characters. An equivalence class is a set of collating elements that all sort to the same primary location. They are enclosed within the bracket-equal delimiters `[=` and `=]`. An equivalence class generally is designed to deal with primary-secondary sorting; that is, for languages like French that define groups of characters as sorting to the same primary location, and then have a tie-breaking, secondary sort. For example, if `e`, `è`, and `ê` belong to the same equivalence class, then `[[=e=]fg`, `[[=è=]fg]`, and `[[=ê=]fg]` are each equivalent to `[eèêfg]`. For more information on collating sequences and their use, see the discussion on using internationalization features in *Command and Shell User's Guide*.

1.2 Using the grep Commands

This section describes the `grep` family of pattern-matching commands (`grep`, `egrep`, and `fgrep`). The differences between these commands are summarized in Table 1-2.

Table 1-2: Versions of the grep Command

grep Version	Description
`grep`	Patterns can contain a limited set of REs. (See the list immediately following this table for exclusions.) The `grep` command uses a compact algorithm that is fast and requires a minimum of program space.
`egrep`	"Extended `grep`" patterns make use of all the REs except the hold delimiters `\(` and `\)`. The `egrep` command uses a deterministic algorithm that requires exponential space.
`fgrep`	"Fixed `grep`" patterns are fixed strings; RE operators are interpreted literally. The `fgrep` command is extremely fast and compact.

The set of REs supported by `grep` is limited so that `grep` can work more efficiently for most uses. The `egrep` command supports the full range of REs shown in Table 1-1, including the following features that are not supported by `grep`:

- The plus sign (`+`)
- The question mark (`?`)

- The vertical bar (|)
- Parentheses (( ))

The `fgrep` command does not allow REs, but it does allow you to specify more than one string. Surround the strings with apostrophes, and separate the strings with newline characters, as in this example using the Bourne shell:

```
$ strings hpcalc | fgrep 'math.h
> fatal.h'
```

In the C shell, you must enter a backslash before each newline character:

```
% strings hpcalc | fgrep 'math.h\
fatal.h'
```

By default, the `grep` commands find each line satisfying the RE or REs you specify. Table 1-3 describes command-line options that allow you to specify other results from your searches.

Table 1-3: Options for the grep Utilities

Option	grep Versions	Description
`-b`	All	Precedes each output line with its disk block number. This option is of use primarily to programmers who are trying to identify specific blocks on a disk by searching for the information contained in them.
`-c`	All	Counts matching lines and prints only the count.
`-e` *`expr`*	All	Uses *`expr`* as the pattern. Useful if *`expr`* begins with a minus sign (–).
`-f` *`file`*	`egrep`, `fgrep`	Uses the contents of *`file`* to supply the expressions to be matched. Specify one expression per line in *`file`*.
`-h`	`egrep`, `fgrep`	Suppresses reporting of file names when multiple files are processed.
`-i`	`grep`, `egrep`	Performs a case-insensitive search.
`-l`	All	Lists only the names of files containing matching lines. Each file name is listed only once, even if the file contains multiple matches. If standard input is specified among the files to be processed with this option, `grep` continues processing; `egrep` and `fgrep` both exit with nonzero status.
`-n`	All	Precedes each matching line with its line number.

Table 1-3: (continued)

Option	grep Versions	Description
`-p expr`	`grep`	Uses *`expr`* as a paragraph separator, and displays the entire paragraph containing each matched line. Does not display the paragraph separator lines. The default paragraph separator is a blank line.
`-q`	`grep`	Operates in ''quiet'' mode, printing nothing except error messages. Useful in shell scripts; exit status reports the success or failure of the search.
`-s`	`grep`	Operates in ''silent'' mode, printing nothing (not even error messages). Useful in shell scripts; exit status reports the success or failure of the search.
`-v`	All	Outputs only lines that do *not* match the specified expressions.
`-w expr`	`grep`	Matches only if *`expr`* is found as a separate word in the text. A word is any string of alphanumeric characters (letters, numbers, and underscores) delimited by nonalphanumeric characters (punctuation or white space.) For example, `word1` is a word; `A+B` is not a word.
`-x`	`fgrep`	Outputs only lines matched in their entirety. Provides a more efficient way to search for REs consisting of `^`*`fixed-string`*`$`.
`-y`	`grep, egrep`	Exact synonym for `-i`.

Matching Patterns and Processing Information with awk 2

This chapter describes the awk command, a tool with the ability to match text on lines in a file and a set of commands that you can use to manipulate the matched lines. In addition to matching text with all of the regular expression (RE) building features that egrep uses, awk treats each line, or **record**, as a set of elements, or **fields**, that can be manipulated individually or in combination. Thus, awk can perform more complex operations, such as:

- Writing selected fields of a record
- Reordering or replacing the contents of a record; for example, to change syntax in a program source file or change system calls when porting from one system to another
- Processing input to find numeric counts, sums, or subtotals
- Verifying that a given field contains only numeric information
- Checking to see that delimiters are balanced in a programming file
- Processing data contained in fields within records
- Changing data from one program into a form that can be used by a different program

The DEC OSF/1 operating system provides several versions of the awk utility:

- The awk command invokes a version that provides the minimum subset of awk features. Programs written using only these features are compatible with the versions of awk on virtually all UNIX platforms.
- The gawk command invokes an enhanced version that offers many additional features. Some or all of these features might or might not be present in other systems' versions of awk; thus, programs using these features might present portability problems.
- The nawk command invokes an enhanced version that is similar to gawk; in addition, nawk is XPG4 conformant.

This chapter describes the awk and gawk versions. Descriptions of features that apply to both of these versions are not marked in any special way. Features available to to gawk but not awk are so indicated. Unless

otherwise indicated, the name awk refers to both versions. For information about unique features of nawk, see the nawk(1) reference page.

2.1 Running the awk Program

The syntax for the awk command is:

awk [**–F***char*] [**–f** *program*] [*file1* [*file2...*]]

The syntax for the gawk command is:

gawk [**–W** *gawk-options*] [**–F***char*] [**–v** *var*=*val*] [**–f** *program*] [**––**] \
[*file1* [*file2...*]]

Table 2-1 describes the options for the awk and gawk commands.

Table 2-1: Options for the awk and gawk Commands

Option	Description
–W *gawk-options* (gawk only)	Specifies actions unique to gawk, such as the use of POSIX compatibility mode. See the gawk(1) reference page for a list of applicable gawk options.
–F*char*	Specifies a character to be used as a field separator. By default, awk uses white space (tabs or spaces) to separate fields in a record. To specify a tab or a shell metacharacter, enclose the entire option in apostrophes. For example: **% awk '-F**Tab**' report** For awk, the –F option must precede any other command-line argument. For gawk, the –W option can precede the -F option.
–v *var=val* (gawk only)	Assigns the value *val* to a variable named *var*; such assignments are available to the BEGIN block of a program.
–f *program*	Specifies the name of a file containing an awk program. This option requires a file name as an argument. The gawk command accepts multiple –f options, concatenating all the program files and treating them as a single program.
–– (gawk only)	Signals the end of options, allowing further arguments to begin with minus signs (–).

Usually, you create an awk program file before running awk. The program file is a series of statements that look like the following:

```
pattern { action }
```

In this structure, a *pattern* is one or more REs that define the text to be matched. Patterns can consist of the following:

- `BEGIN` or `END`
- Boolean combinations of REs using the operators `!` (NOT), `||` (Logical OR), and `&&` (AND), with parentheses for grouping expressions
- Boolean combinations of relational operations on strings, numbers, fields, and variables
- Ranges of records, specified in this way:

  ```
  pattern1,pattern2
  ```

- In `gawk` only, conditional expressions (see Table 2-3 for an explanation of the conditional operator)

An *action* is one or more steps to be executed, designated with `awk` commands, operands, and operators. Actions can consist of the following:

- Assignment statements
- Statements to format and print data
- Tests to alter the flow of control
- Control structures, such as `if-else`, `while`, and `for` statements
- Redirection of output to one or more output streams besides standard output
- Piping of output (and, in `gawk`, input)

The braces (`{ }`) are delimiters separating the action from the search pattern. Actions can be specified on a single line, or multiple lines to give a visual structure to the program. If you place an action consisting of several commands on one line, separate the commands with semicolons (`;`). For example, either of the two following programs will find every record matching either "Gunther" or "gunther". For each matching record, it will print two lines, first the number of the record on which the match was made and then the first two fields of the matched record:

Program 1:

```
/[Gg]unther/ { print "Record:", NR ; print $1, $2 }
```

Program 2:

```
/[Gg]unther/ {
  print "Record:", NR
  print $1, $2
}
```

Output from this program might look like the following:

```
Record: 382
Schuller Gunther
Record: 397
schwarz gunther
```

Both the pattern and the action are optional elements of a program line. If you omit the pattern, awk performs the action on every record in the file; if you omit the action, awk copies the record to standard output. A null program passes its input unmodified to the output.

Once you create the program file, enter the awk command on the command line as follows:

```
$ awk -f progfile infile > outfile
```

This command uses the program in `progfile` to process `infile`, and writes the output to `outfile`. The input file is not changed.

With a short program, you can accomplish the same job by entering the program on the command line before the name of the input file. For example:

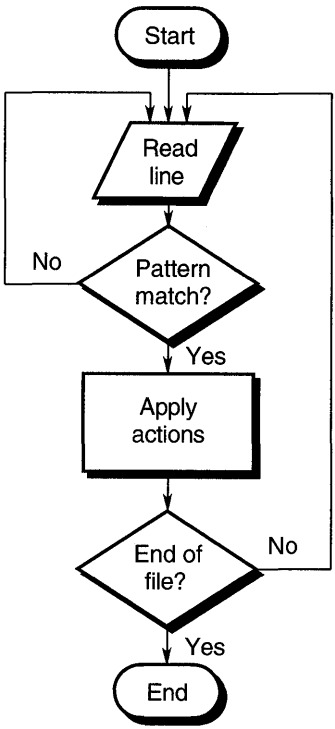
```
$ awk '/[Gg]unther/ { print $1, $2 }' infile
```

When you use awk in this way, enclose the program in single or double quotation marks as required to control shell file name expansion and variable substitution.

When you start awk, it reads the program, checking for syntax. It then reads the first record of the input file, testing the record against each of the patterns in the program file in order of their appearance. When awk finds a pattern that matches the record, it performs the associated action. Then awk continues to search for matches in the program file. When it has compared the first input record against all patterns in the program file and performed all the actions required for that record, awk reads the next input record and repeats the program with that record. Processing continues in this manner until the end of the input file is reached. Figure 2-1 is a flowchart of this

sequence. Compare the operation of awk with the very similar operation of the sed editor, shown in Figure 3-1.

Figure 2-1: Sequence of awk Processing



2.2 Printing in awk

You can use either the `print` command or the `printf` command to produce output in awk. The `print` command prints its arguments as already described; that is, arguments separated by commas are printed separated by the current output field separator (OFS), and arguments not separated by commas are concatenated as they are printed.

The `printf` command has a syntax identical to that of the `printf` statement in the C programming language:

printf(*f*,*e1*[,*e2*,...])

This command prints the arguments *e1* and so on, formatted as defined by *f*.

Refer to the gawk(1) and printf(3) reference pages for information on constructing format specifiers.

2.3 Using Variables in awk

You can create and operate on variables in an awk program. For example, the following assignment statement creates a variable named var whose value is the sum of the third and fourth fields in the record:

```
var = $3 + $4
```

You can use variables as part of a pattern, and you can manipulate them in actions. For example, the following program assigns a value to a variable named tst and then uses tst as part of a pattern for further actions:

```
{ tst = $1 }
tst == $3 { print }
```

The awk program recognizes the set of built-in variables listed in Table 2-2.

Table 2-2: Built-In Variables in awk

Variable	Description
$0	The contents of the current record.
$*n*	The contents of field *n* of the input record.
ARGC (gawk only)	A count of the arguments given to awk. This variable is modifiable. Does not include the command name, options preceded by minus signs, the script file name (if any), or variable assignments.
ARGV (gawk only)	An array containing the arguments given to awk. The elements of this array are modifiable. Does not include the command name, options preceded by minus signs, the script file name (if any), or variable assignments.
CONVFMT (gawk only)	The conversion format for numbers (by default, %.6g).
ENVIRON (gawk only)	A modifiable array containing the current set of environment variables; accessible by ENVIRON[*var*], where *var* is the name of the environmental variable. Changing an element in this array does not affect the environment passed to commands that gawk spawns by redirection, piping, or the system() function.
FIELDWIDTHS (gawk only)	A white-space separated list of field widths. When this variable is set, gawk parses input records into fields of fixed widths, ignoring the FS variable.

Table 2-2: (continued)

Variable	Description
FILENAME	The name of the current input file. If no input file was named, FILENAME contains a single minus sign.
FNR (gawk only)	The number of the current record within the current file. Differs from NR if multiple files are being processed and the current file is not the first file read.
FS	The character or expression used for a field separator. By default, any amount of white space. In awk, the FS variable is modifiable to any single character. In gawk, field separators can be multibyte REs and can be multiply defined. For example, the following statement defines either a comma followed by any amount of white space or at least one white-space character as the field separator: FS = ",[Tab]*|[Tab]+"
IGNORECASE (gawk only)	Switch for case sensitivity in RE matching. If IGNORECASE is nonzero, REs are case insensitive; for example, the RE /aB/ matches "ab" or "Ab" or "aB" or "AB". The default value is 0 so that REs are case sensitive.
NF	The number of fields in the current record.
NR	The number of the current record, counted sequentially from the beginning of the first file read. Differs from FNR if multiple files are being processed and the current file is not the first file read.
OFMT	The format specification for numbers on output (by default, %.6g).
OFS	The output field separator; the character (or, for gawk, the string inserted between fields when the data is written). By default, a space character.
ORS	The character used for the output record separator (the character between records when the data is written). By default, a newline character.
RLENGTH (gawk only)	The length of the string matched by match(); set to –1 if no match.
RS	The character used for a record separator.
RSTART (gawk only)	The index (position within the string) of the first character matched by match(); set to 0 if no match.

Table 2-2: (continued)

Variable	Description
SUBSEP (gawk only)	The separator for multiple subscripts in array elements (by default \034, the ASCII FS character). (See the gawk(1) reference page for more information.)

2.4 Performing Actions Before or After Processing the Input

The awk program recognizes two special pattern keywords that define the beginning (BEGIN) and the end (END) of the input file. BEGIN matches the beginning of the input before reading the first record. Therefore, awk performs any actions associated with this pattern once, before processing the input file. BEGIN must be the first pattern in the program file. For example, to change the field separator to a colon (:) for all records in the file, include the following line as the first line of the program file:

```
BEGIN { FS = ":" }
```

This example action works the same as using the –F: option on the command line.

Similarly, END matches the end of the input file after processing the last record. Therefore, awk performs any actions associated with this pattern once, after processing the input file. END must be the last pattern in the program file. For example, to print the total number of records in the input file, include the following line as the last line in the program file:

```
END { print NR }
```

2.5 Using Regular Expressions as Patterns

The simplest RE is a literal string of characters. REs in awk must be enclosed in slashes. To include a slash as part of an RE, escape the slash with a backslash. For example, /\/usr\/share/ is an RE that matches the string /usr/share. The following example shows a complete awk program that will print every record containing the string "the":

```
/the/
```

Because this RE does not specify blanks or other qualifiers, the program displays records containing "the" as a separate word and records containing the string as part of words such as "northern". REs are case sensitive. To find either "The" or "the", use a bracketed RE as follows:

```
/[Tt]he/
```

The awk program uses the same set of REs that egrep uses. See Table 1-1 for a list of the RE elements that awk supports. In awk, the circumflex (^) and dollar sign ($) can apply to a specific field or variable as well as to the entire line. The following example will match a field consisting of the word "cat" or the word "cats" but will not match any word containing these strings (such as "concatenate"):

```
{ for (i=1;i<=NF;i++) if ($i ~ /^cats?$/) print }
```

2.6 Using Relational Expressions and Combined Expressions as Patterns

Several examples in previous sections have illustrated the use of relational expressions in patterns. Relational expressions allow you to restrict a match to a specific field of a record or to make other tests, either numeric or string-related. The awk program defines the following relational operators for use in building patterns:

==	Equivalent
!=	Not equivalent
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
~	Matches RE
!~	Does not match RE

Use the == (equivalent) and != (not equivalent) operators to test literal strings and numeric values. For example:

```
str == "literal string"
num != 23
$NF == 1991
```

The last line in this example uses the $*n* syntax combined with the built-in variable NF to test the value of the last field of a record. To test against REs, use the ~ (matches RE) and !~ (does not match RE) operators as follows:

```
str ~ /[Ll]iteral/
```

Relational expressions can be tested against built-up expressions. For example, the following pattern finds all records whose second field (`$2`) is at least 100 greater than the first field (`$1`):

```
$2 > $1 + 100
```

The following pattern finds records that contain an even number of fields:

```
NF % 2 == 0
```

Use the operators listed in Section 2.11 to build expressions.

You can use magnitude-comparison operators to test strings. For example, the following pattern finds records that begin with ''s'' or any character that appears after it to the end of the character set:

```
$0 >= "s"
```

You can combine two or more patterns by using the following Boolean operators:

`&&`	AND
`||`	Logical OR
`!`	NOT

For example, to prevent nonalphanumeric matches in the preceding example, you can combine two expressions as follows:

```
($0 >= "s" && $0 < "{")
```

(The left brace is the character immediately following the letter ''z'' in the ASCII code.)

2.7 Using Pattern Ranges

You can use a pattern range to select a group of records to operate on. A pattern range consists of two patterns separated by a comma; the first pattern specifies the start of the range, and the second pattern specifies the end of the range. The `awk` program performs the associated action on all records in the range, including the records that match the two patterns. For example:

```
NR==100,NR==200 { print }
```

This program prints 101 records from the input file, beginning with record 100 and ending with record 200.

Using a pattern range does not disable other patterns from matching records within the range. However, because the input file is processed record by record, with each record being subject to all the actions appropriate to it before the next record is considered, the actions taken can appear to be out of sequence. For example:

```
2,4 { print }
/share/ { print "Found share" }
```

Apply this program to the following input file:

```
This is a test file
Line two
Try to share things
Line four
Last line of file
```

When this file is processed by awk, the output is as follows:

```
Line two
Try to share things
Found share
Line four
```

The second action is applied to record 3 before record 4 is examined to see if it matches the first pattern.

2.8 Actions in awk

An action can be a single command, such as `print`, or it can be a series of commands. An action can include tests to select records or parts of records; if desired, you can create a program that has no explicit patterns, relying instead on relational comparisons within its actions. Such a program can bear a strong resemblance to a C program; for example:

```
{
  if ($1 == 0) {
    print;
    printf("%5.2f\n", $2+$3)
  } else {
    printf("%5.2f\n", $1+$2)
  }
}
```

Note

The semicolon after the `print` command, which would be required in a C program, is not required by awk; but it does not cause an error.

2.9 Concatenating Strings

You concatenate strings by placing their variable names together in an expression. For example, the command `print $1 $2` prints a string consisting of the first two fields from the current record, with no space between them. You can use variables, numeric operators, and functions when concatenating strings. (See Section 2.10.1 and Section 2.11 for

information on variables and numeric operators.) The function `length($1 $2 $3)` returns the length in characters of the first three fields. (See Section 2.12 for a list of the functions in `awk`.) If the strings you want to concatenate are field variables (see Section 2.10.2), you are not required to separate the names with white space; the expression `$1$2` is identical to `$1 $2`.

2.10 Using Variables in an Action

The `awk` program uses variables to manipulate information. Variables are of the following three types:

- Simple variables
- Field variables
- Array variables

This section discusses these types of variables and how to use them.

2.10.1 Simple Variables

You can create any number of simple variables, assigning values to them as required. If you refer to a variable before explicitly assigning a value to it, `awk` creates the variable and assigns it a value of 0 (zero). Variables can have numeric (floating-point) values or string values depending on their use in the action expression. For example, in the expression `x = 1`, *x* is a numeric variable. Similarly, in the expression `x = "smith"`, *x* is a string variable. However, `awk` converts freely between strings and numbers when needed. Therefore, in the expression `x = "3"+"4"`, `awk` assigns a value of 7 (numeric) to *x*, even though the arguments are literal strings. If you use a variable containing a nonnumeric value in a numeric expression, `awk` assigns it a numeric value of 0. For example:

```
y = 0
z = "ABC"
x = y+z
print x, z
```

This sequence prints ''0 0'' because *y* is assigned a value of 0 and *z* assumes a value of 0 when used numerically.

You can force a variable to be treated as a string by concatenating the null string (`""`) to the variable; for example, `x = 2 ""`. (See Section 2.9 for information on concatenating strings.) You can force a variable to be treated numerically by adding zero to it. Forcing variables to be treated as particular types can be useful. For example, if *x* is ''0100'' and *y* is ''1'', `awk` normally treats both variables as numerics and considers that *x* is greater than *y*. Forcing both variables to be treated as strings causes *x* to be less than *y* because ''0'' precedes ''1'' in the ASCII code.

2.10.2 Field Variables

Fields in the current record, also called **field variables**, share the properties of simple variables. They can be used in arithmetic or string operations and can be assigned numeric or string values. (You cannot modify the contents of the current record (`$0`) explicitly, but you can alter it indirectly by modifying all of the individual fields.) The following action replaces the first field with the record number and then prints the resulting record:

```
{ $1 = NR; print }
```

The next example adds the second and third fields and stores the result in the first field:

```
{ $1 = $2 + $3; print $0 }
```

(Printing `$0` is identical to printing with no arguments.)

You can use numeric expressions for field references; the following example prints the first, second, and sixth fields:

```
i = 1
n = 5
{ print $i, $(i+1), $(i+n) }
```

As described in Section 2.10.1, `awk` converts between string and numeric values. How you use a field determines whether `awk` treats it as a string or numeric value. If it cannot tell how a given field is used, `awk` treats it as a string.

The `awk` program splits input records into fields as needed. You can split any literal string or string variable into an array by using the `split` function. For example:

```
x = split(s, array1)
y = split("Thu Dec 18 11:19:40 EST 1992", array2)
```

The first line in this example splits the variable *s* into elements of an array named `array1`, creating `array1[1]` to `array1[`*n*`]` where *n* is the number of fields in the string. The second line splits a literal string in the same manner into `array2`. The `split` function can split strings by using an alternative field separator; see Section 2.12 for more information on using this function. See Section 2.10.3 for information on using arrays.

2.10.3 Array Variables

Like field variables, array variables share the properties of simple variables. They can be used in arithmetic or string operations and can be assigned numeric or string values. You do not need to declare or initialize array elements; `awk` creates them and initializes them to zero upon first reference. Subscripts are indicated by being enclosed in brackets. You can use any value that is not null, including a string value, for a subscript. An example

of a numeric subscript follows:

```
x[NR] = $0
```

This expression creates the *NR*th element of the array *x* and assigns the contents of the current input record to it. The following example illustrates using string subscripts:

```
/apple/  { x["apple"]++ }
/orange/ { x["orange"]++ }
END      { print x["apple"], x["orange"] }
```

For each input record containing `apple`, this program increments the *appleth* element of array *x* (and similarly for `orange`), thereby producing and printing a count of the records containing each of these words.

Problems can occur when you use an `if` or `while` statement to locate an array element. If the array subscript does not exist, the statement adds the subscript as a new B-tree node with a null value. For example:

```
{ if (exists[$2] == 1) }
```

To avoid this type of problem, use code similar to the following, in which *i* is printed after an `if` statement within a `for` loop:

```
for (i in exists) {
  if (exists[i] != "") print i
}
```

Also use this type of coding when `while` is used with a relational operator.

2.11 Using Operators in an Action

Use the operators shown in Table 2-3 to build expressions within the action statement.

Table 2-3: Operators for awk Actions

Operator	Description	Example
+	Addition	2+3 = 5
+	(gawk only) Unary plus; placeholder	+4 = 4
-	Subtraction	7-3 = 4
-	(gawk only) Unary minus	-4 is negative 4
*	Multiplication	2*4 = 8
/	Division	6/3 = 2
%	Modulo (remaindering)	7%3 = 1

Table 2-3: (continued)

Operator	Description	Example
++	Increment	See the description following this table.
--	Decrement	See the description following this table.
+=	Increment by value	x+=y is equivalent to x = x+y
-=	Decrement by value	x-=y is equivalent to x = x–y
*=	Multiply by value	x*=y is equivalent to x = x*y
/=	Divide by value	x/=y is equivalent to x = x/y
%=	Modulo by value	x%=y is equivalent to x = x%y
?...:	(gawk only) Conditional	See the description following this table.

The following example prints the sum of all the first fields and the sum of all the second fields in the input file:

```
{ s1 += $1; s2 += $2 }
END { print s1,s2 }
```

The position of the increment and decrement operators affects their interpretation. The expression `i++` evaluates the current contents of `i` and then increments `i`. The expression `++i` causes `awk` to increment `i` before evaluation. For example:

```
$ echo "3 3" | awk '{
>   print "$1 =", $1 "; $1++ =", $1++ "; new $1 =", $1
>   print "$2 =", $2 "; ++$2 =", ++$2 "; new $2 =", $2
> }'
$1 = 3; $1++ = 3; new $1 = 4
$2 = 3; ++$2 = 4; new $2 = 4
```

The conditional operator is used in the following form:

expr`?`*expr1*`:`*expr2*

This structure returns the value of *expr1* if *expr* is nonzero; otherwise, it returns the value of *expr2*. For example, `x=(3-1)?4:5` returns 4, while `x=(3-3)?4:5` returns 5.

2.12 Using Functions Within an Action

The `awk` language includes the built-in functions listed in Table 2-4. Additionally, `gawk` allows you to create additional functions as described after the table.

Table 2-4: Built-In awk Functions

Function	Description
`atan(expr)` (gawk only)	Returns the arctangent of the value specified by *expr*.
`close(arg)` (gawk only)	Closes the file or pipe named by `arg`.
`cos(expr)` (gawk only)	Returns the cosine of the value specified by *expr*.
`delete(array[sub])` (gawk only)	Deletes the element of *array* indicated by *sub*.
`exp(arg)`	Returns the natural antilogarithm (base ε) of `arg`. For example, `exp(0.693147)` returns 2. See `log(arg)`.
`gsub(expr,s1,s2)` (gawk only)	Replaces every sequence of characters in string *s2* that matches the RE *expr* with the string specified by *s1*. If *s2* is not supplied, the current input record is used. Expression *expr* is reevaluated for each match. This function returns a value representing the number of replacements. See `sub(expr,s1,s2)`.
`index(s1,s2)`	Returns the character position in string *s1* where string *s2* occurs. If *s2* is not in *s1*, this function returns a zero.
`int(arg)`	Returns the integer part of *arg*.
`length`	Returns the length in characters of the current record.
`length(arg)`	Returns the length in characters of the string specified by *arg*. See `length`.
`log(arg)`	Returns the natural logarithm (base ε) of *arg*. For example, `log(2)` returns 0.693147. See `exp(arg)`.
`match(s,expr)` (gawk only)	Returns the character position in string *s* where a match is found for the RE *expr*; sets the variable `RSTART` to the character position at which the match begins and `RLENGTH` to a value representing the length of the matched string. If no match is found, this function returns a zero.
`rand` (gawk only)	Returns a pseudorandom number ($0 \leq n < 1$).

Table 2-4: (continued)

Function	Description
`split(s,array,sep)`	Splits string *s* into consecutive elements of *array*`[1]...[n]` and returns the number of elements. The optional *sep* argument specifies a field separator other than the one currently in force (the contents of the `FS` variable).
`sprintf(f,e1,e2,...)`	Returns (but does not print) a string containing the arguments *e1* and so on, formatted in the same manner as by the `printf` command.
`sqrt(arg)`	Returns the square root of *arg*.
`sin(arg)` (gawk only)	Returns the sine of *arg*.
`srand(seed)` (gawk only)	Uses *seed* as the seed for a pseudorandom number sequence for subsequent calls to `rand`. If no seed is specified, the time of day is used. The return value is the previous seed.
`strftime(f,time)` (gawk only)	Formats *time* as specified by *f*. The *time* value should be specified in the form returned by `systime()`. See the `strftime`(3) reference page for a list of the format conversions that are available.
`sub(expr,s1,s2)` (gawk only)	Replaces the first sequence of characters in string *s2* that matches the RE *expr* with the string specified by *s1*. If *s2* is not supplied, the current input record is used. This function returns a value representing the number of replacements (0 or 1). See `gsub(expr,s1,s2)`.
`substr(s,m,n)`	Returns the substring of *s* that begins at character position *m* and is *n* characters long. The first character in *s* is at position 1. If *n* is omitted or if the string is not long enough to supply *n* characters, the rest of the string is returned.
`system("command")` (gawk only)	Executes the system command specified and returns its exit status. The entire command must be enclosed in quotation marks to prevent `gawk` from attempting to interpret it as one or more variable names.
`systime()`	Returns the current time of day as the number of seconds since midnight, January 1, 1970.
`tolower(s)` (gawk only)	Translates all uppercase letters in string *s* to lowercase. If there is no argument, the function operates on the current record.

Table 2-4: (continued)

Function	Description
`toupper(`*s*`)` (`gawk` only)	Translates all lowercase letters in string *s* to uppercase. If there is no argument, the function operates on the current record.

The `gawk` language also allows you to create functions by using the following syntax:

function *name*(*parameter-list*) **{**
    statements
}

The word `func` can be used in place of `function`. For functions that you create, the left parenthesis both in the function's definition and in its use must immediately follow the function name, with no intervening space. The names in the function declaration's parameter list are the formal parameters for use within the function. When you call a function, `gawk` replaces these formal parameters with the values you supply in the calling statement. Functions can be recursive.

You can define local variables for a given function by declaring them as extra formal parameters; upon function entry, all local variables are initialized as empty strings or the number 0. To avoid visual confusion between real parameters and local variables, you can separate the local variables with extra spaces in the function declaration. For example:

```
function foo(in, out,      local1, local2) {
  local1 = "foo"
  local2 = "bar"
   •
   •
   •
}
```

2.13 Using Control Structures in awk

The `awk` language provides the control structures listed in Table 2-5. Except where noted, these structures work exactly as they do in the C language. To perform several statements in a single control structure's action, enclose the statements in braces. If only a single statement is to be performed, the braces are optional. Each of the first two `if` structures in the following example includes a single statement to be executed; these structures are equivalent:

```
{
  if (x == y) print
  if (x == y) {
    print
  }
  if (x == y) {
    print $3
    printf("Sum = %d\n", x+z)
  }
}
```

Table 2-5: Control Structures in awk

Structure	Description
`if-else`	The condition in parentheses in an `if-else` structure is evaluated. If it is true, the statements following the `if` are performed. If it is false, the statements following the optional `else` keyword (if present) are performed.

`while`	The statements following the `while` statement are performed until the tested condition is not true. The following example prints all the fields in the input records, one field per line:
	`{`
	`  i = 1`
	`  while(i<=NF) print $i++`
	`}`

`for`	The `for(`*expr1*`;`*expr2*`;`*expr3*`) statements` structure is equivalent to the following `while` construct:
	`{`
	`  `*expr1*
	`  while(`*expr2*`) {`
	`    `*statements*
	`    `*expr3*
	`  }`
	`}`
	The previous `while` example could also be written as follows:
	`{`
	`  for(i=1;i<=NF;++i) print $i`
	`}`

`break`	The `break` statement causes an immediate exit from an enclosing `while` or `for` loop.
Comments	Include comments in an `awk` program file to explain program logic. Comments begin with the number sign (`#`) and end with the end of the line. For example:

Table 2-5: (continued)

Structure	Description

	`{`
	`  print x,y  # This is a comment`
	`}`

`continue`	The `continue` statement causes the next iteration of an enclosing loop to begin.
`getline`	The `getline` statement causes `awk` to discard the current input record, read the next input record, and continue scanning patterns from the present location in the program file. Contrast this behavior with that of the `next` statement.
`next`	The `next` statement causes `awk` to discard the current input record, read the next input record, and begin scanning patterns from the start of the program file. Contrast this behavior with that of the `getline` statement.
`exit`	The `exit` statement causes the program to stop as if the end of the input occurred.

2.14 Redirection and Pipes

Unless otherwise specified, `print` and `printf` statements write their output to the standard output file. You can redirect the output of any printing statement by using standard redirection operators. For example:

```
print $0, $3, amt >> "reportfile"
```

This example appends its output to a file named `reportfile` instead of writing to the standard output. (If `reportfile` does not exist before the first instance of redirection, it is created.) Note that the output file name in this example is enclosed in quotation marks. The quotation marks are required to distinguish the file name from a variable name. You can mix writing to named files with writing to the standard output.

You can also pipe printed output through other commands. The following example pipes `awk`'s output through the `tr` command to convert all uppercase letters to lowercase letters:

```
print | "tr [A-Z] [a-z]"
```

As with redirection, the command to which you pipe the output must be enclosed in quotation marks. In `gawk`, you can also pipe input to a `getline` statement.

Only a limited number of files can be open for output. For `awk`, this limit is 10 files. The `gawk` program uses your default open file descriptor limit. For efficiency, however, you can use the `close(`*`arg`*`)` statement to close files that you have opened for output and no longer need.

Editing Files with the sed Editor 3

You do not need to know how to use the ed line editor to use the material presented here, although the sed stream editor is a program that works much like ed. Unlike ed, however, sed edits files by using a prepared list of commands, called a **script**, instead of interacting with the user. This method of operation makes sed particularly well suited for tasks like the following:

- Editing large files
- Performing complex editing operations many times without extensive retyping and cursor positioning
- Performing global changes in one pass through the input

The sed stream editor receives its input from standard input or from a named file, changes that input as directed by commands in a command file or on the command line, and writes the resulting stream to standard output. If you specify more than one input file, sed processes each file in sequence and concatenates the results to standard output. If you do not provide a command file and do not use any with the sed command options, sed copies standard input to standard output without change. The editor keeps only a few lines of the file being edited in memory at one time and does not use temporary files. Therefore, the size of the file to be edited is limited only by the available disk space.

The command script for sed can be a file that you create before running the editor, a series of commands you enter as a command option, or both. The editor cannot process more than 99 commands in a single invocation; for this reason or to accomplish certain extremely complex editing tasks, you might need to pipe the output from sed into another instance of sed.

3.1 Running the sed Editor

The syntax for the sed command is as follows:

sed [**-n**] [**-e** *commands*] [**-f** *script*] [*source_file1* [*source-file2...*]]

Table 3-1 describes the options for the sed command.

Table 3-1: Options for the sed Command

Option	Description
–e *commands*	Specifies editing commands on the command line. This option requires an argument (*commands*) consisting of valid `sed` editing commands. Enclose the argument in single or double quotation marks as required to control shell file name expansion and variable substitution.
–f *script*	Specifies a file containing a prepared script of editing commands. This option requires a file name as an argument.
–n	Inhibits normal writing of edited lines. When this option is used, only lines explicitly printed with the `sed` editor's `p` and `P` commands are written to standard output.

Usually, you create a command file containing the desired editing commands before running `sed`. The `sed` editor's command set is powerful and requires little typing. Each command in the command file can be on a separate line, or you can place multiple commands on one line by separating them with semicolons (`;`). For example, either of the following two scripts will delete all lines beginning with `.ne`, `.RE`, or `.RS`:

Script 1:

```
/^\.ne/d
/^\.R[ES]/d
```

Script 2:

```
/^\.ne/d;/^\.R[ES]/d
```

Once you create the command file (`cmdfile` in the following example), enter the `sed` command as in this example:

```
$ sed -f cmdfile infile > outfile
```

This command edits `infile`, using the commands contained in `cmdfile` and writing the output to `outfile`. The input file is not changed.

With a short editing script, you can accomplish the same job by entering the editing commands as an argument to the –e option on the command line:

```
$ sed -e '/^\.ne/d;/^\.R[ES]/d' infile > outfile
```

If you use the –e and –f options together on a command line, `sed` applies all the commands specified by both options, in the order in which the options appear on the command line. For example:

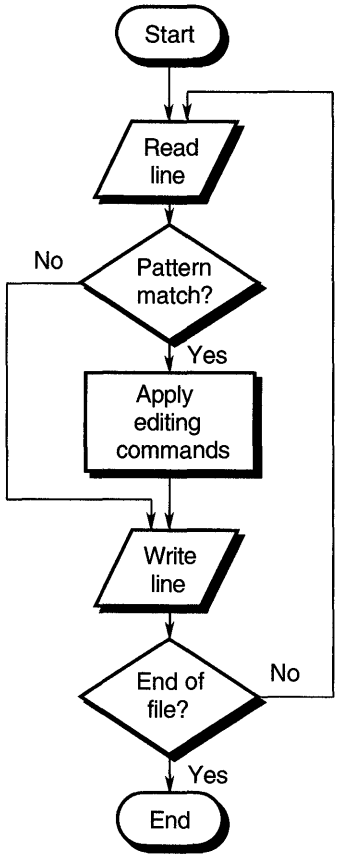
```
$ echo "s/line/foo/" > sedx
$ echo "Test line" | sed -f sedx -e 's/line/bar/'
Test foo
$ echo "Test line" | sed -e 's/line/bar/' -f sedx
Test bar
```

You can use the –e and –f options more than once with a given sed command. For example:

```
$ sed -f script1 -e 's/foo/bar/' -f script2 msgs > msgs2
```

When you start sed, the editor reads and compiles the command script, checking for syntax and organizing the commands for efficiency. It then reads the input file one line at a time into an area of memory called the **pattern space**. The editor then tries to match the addresses specified by the commands in the script, one after another, to the lines in the pattern space. Whenever a command's address matches any line or lines in the pattern space, sed applies that editing command to the matched text. Commands are applied in sequence to the text, and the results of each command are used as the input for subsequent commands. When no more commands match a given line in the pattern space, sed writes that line to the output, reads more input, and repeats the process. Figure 3-1 is a flowchart of this sequence. Compare the operation of sed with the very similar operation of the awk program, shown in Figure 2-1.

Figure 3-1: Sequence of sed Processing



Some editing commands change the way the editing process operates by causing the editor to bypass other script commands, by inhibiting the writing of certain lines (by deleting them), or by ending the process prematurely.

3.2 Selecting Lines for Editing

The `sed` editor identifies lines to be edited by matching addresses. An address can be either a line number or a context address:

- Line numbers

  The first line in the input stream is line 1, and each successive line increments the line counter by one. The dollar sign ($) is a shorthand way to specify the last line of the input stream.

If you edit more than one file in a single invocation of `sed`, the line counter is cumulative across all the files edited; for example, if the first file contains 100 lines, the first line of the second file is line 101.

- Context addresses

  A context address is a regular expression (RE) enclosed in slashes; for example, `/^\.R[ES]/` matches any line beginning with either `.RE` or `.RS`.

The `sed` editor recognizes the limited set of REs shown in Table 3-2.

Table 3-2: Regular Expressions Recognized by sed

Expression	Name	Rule
	Period (dot)	Matches any single character except the newline character.
`\n`	Embedded newline (a backslash followed by the letter n)	Matches an embedded newline character in a line formed by joining multiple lines.
`*`	Asterisk	Matches any number of occurrences of the preceding simple RE, including none.
`[`*chars*`]`	Brackets	Matches a single instance of any one of the characters within the brackets. Ranges of characters can be abbreviated by using a dash; for example, `[0-9a-z]` matches any single digit or lowercase letter.
`^`	Circumflex	When used at the beginning of an RE, matches the beginning of a line. When used as the first character inside brackets, excludes the bracketed characters from being matched. Otherwise, has no special properties.
`$`	Dollar sign	When used at the end of an RE, matches the end of a line. Otherwise, has no special properties.
`\`*char*	Backslash	Escapes the next character to permit matching on explicit instances of characters that are normally RE operators.
`[:`*class*`:]`	Class	A character class name enclosed in bracket-colon delimiters matches any of the set of characters in the named class.

Table 3-2: (continued)

Expression	Name	Rule
\(*expr*\)	Substring delimiters	Matches *expr* and saves the matching substring into a numbered holding space for reuse with the \n operator.
\n	Repeat expression (a backslash followed by a single digit *n*)	Repeats the expression delimited by the nth set of hold delimiters in the RE.
//	Empty slashes	Matches the text that matched the most recently specified RE.

Some sed commands do not accept addresses. Commands that accept addresses behave differently depending on the number of addresses, as follows:

- If no address is specified, the command is applied to every line in the input stream.
- If one address is specified, the command is applied to each line that matches the address.
- If two addresses are specified, the command is applied to a group of lines starting with a line that matches the first address and ending with the first subsequent line that matches the second address. The editor then tries to match the first address again to find another range.

Caution

If two addresses are specified but sed cannot find a line matching the ending address, sed operates on every line from the first address to the end of the file.

3.3 Summary of sed Commands

Each sed command consists of a single letter with optional addresses. Some commands require arguments and accept qualifiers that alter their behavior. Do not include any space between the addresses and the letter. If you use two adddresses with a command, separate them with a comma. The r and w commands and the w flag for the s command require a single space between the letter and the argument; otherwise, do not include any space between the letter and the argument.

Table 3-3, Table 3-4, and Table 3-5 describe the individual sed commands, showing the syntax of each. In these tables, the following conventions apply:

- The term ''range of lines'' can mean a single line, a group of lines, or all lines, as specified by the number of addresses given to the command.
- Brackets [ ] enclose optional elements. Nested brackets indicate that the nested element can be used only if the enclosing element is present.
- Italic (slanted) type indicates a general name for an object that you specify; for example, *file* represents a command argument that must be the name of a file.

The following example illustrates a correctly formed `s` command with all optional elements:

```
1,/^$/s/vizier//g
```

This example processes the header of a mail message (line 1 to the first completely blank line), replacing the string `vizier` with nothing wherever the string occurs on any line in the specified range.

Table 3-3: Text Editing and Movement Commands

Command	Description
Append text	
[*addr1*]a\ *text*[\ *text...*]	Writes the specified text[a] to the output after the line specified by *addr1*. See also the `i` command.
Change lines	
[*addr1*[,*addr2*]]c\ *text*[\ *text...*]	Deletes the addressed range of lines and writes the specified text[a] to the output in its place.
Delete lines	
[*addr1*[,*addr2*]]d	Deletes the specified range of lines.[b]
Delete the first line of the pattern space	
[*addr1*[,*addr2*]]D	Deletes all text in the pattern space up to and including the first newline character. If only one line is in the pattern space, this command reads another line from the input into the pattern space. After these operations, the command starts the complete list of editing commands again from the beginning.

Table 3-3: (continued)

Command	Description
Insert lines	
[*addr1*]i\ *text*[\ *text...*]	Writes the specified text[a] to the output before the line specified by *addr1*. See also the a command.
Advance in the file	
[*addr1*[,*addr2*]]n	Writes the indicated range from the pattern space (if not deleted) to the output and then reads the next line from the input into the pattern space.
Join lines	
[*addr1*[,*addr2*]]N	Joins the indicated lines together as a single line with embedded newline characters. If only one address is given, the command joins the specified line to the next line in the input stream. Pattern matches for addressing or for string replacement can extend across embedded newline characters. Use \n to indicate an embedded newline character for matching.
Print lines	
[*addr1*[,*addr2*]]p	Writes the specified range of lines to the output at the point in the editing process where the p command appears. This command can be used to reorder sections of a file.
Print the first line in the pattern space	
[*addr1*[,*addr2*]]P	Writes all text in the pattern space, up to and including the first newline character, to the output at the point in the editing process where the P command appears.
Read and append a file	
[*addr1*]r *file*	Reads the named file[c] and writes the file's contents to the output after *addr1*.

Table 3-3: (continued)

Command	Description
Substitute text	
[*addr1*[,*addr2*]]s/*expr*/*string*/[*flags*]	Searches the indicated lines for a string of characters matching the RE defined by *expr*, and replaces that set of characters with *string*. This command's operation is modified by the g, p, and w *file* flags.[d] If either *expr* or *string* includes a slash (/), you must escape the literal slash with a backslash (s/path/path\/file/) or use alternative delimiters such as the at sign (@) or question mark (?). For example, s@path@path/file@ replaces path with path/file.
Write a named file	
[*addr1*[,*addr2*]]w *file*	Writes the specified range of lines to the named file at the point in the editing process where the w command appears.[e]
Print line number	
[*addr1*]=	Writes the line number of the indicated line to the output.

Table Notes:

a. If the text to be written consists of multiple lines, each line except the last must have a backslash (\) before the terminal newline character. The text is always written regardless of anything subsequent commands do to the line that caused it to be written, including deletion of that line. It is neither scanned for address matches nor affected by subsequent editing commands, and it has no effect on the editor's line counter.

b. If no addresses are given, the d command deletes all lines in the pattern space; unless constrained by a range controlling a group of commands in braces, the command deletes the entire contents of the file.

c. Include exactly one space between the r command and the file name. If *file* cannot be accessed, sed behaves as if it had read an empty file

and gives no abnormal indication. A combined maximum of 10 files can be named for reading or writing in any given editing process.

d. See Section 3.4 for descriptions of the `s` command's optional flags.

e. Include exactly one space between the `w` command and the file name. If *`file`* exists, it is overwritten; if not, it is created. A combined maximum of 10 files can be named for reading or writing in any given editing process.

Table 3-4: Buffer Manipulation Commands

Command	Description
Retrieve text from hold area	
[*addr1*[,*addr2*]]g and [*addr1*[,*addr2*]]G	Copies the contents of the hold area to the pattern space indicated by *addr1* and *addr2*, if present. The `g` command destroys the existing contents of the pattern space; the `G` command appends the held text to the contents of the pattern space, separating the previous text from the appended text with a newline character.
Move text to the hold area	
[*addr1*[,*addr2*]]h and [*addr1*[,*addr2*]]H	Copies the indicated range from the pattern space to the hold area. The `h` command destroys the existing contents of the hold area; the `H` command appends the text in the pattern space to the contents of the hold area, separating the previous text from the appended text with a newline character.
Exchange pattern space and hold area	
[*addr1*[,*addr2*]]x	Exchanges the contents of the pattern space with those of the hold area.

Table 3-5: Flow-of-Control Commands

Command	Description
Range negation ("Don't")	
[*addr1*[,*addr2*]]!*cmd*	The exclamation point (`!`) instructs `sed` to apply the command following it on the same line to the parts of the input file that are *not* selected by `addr1` and `addr2`.

Table 3-5: (continued)

Command	Description
Command grouping	

`[`*addr1*`[,`*addr2*`]]{`
nested commands
`}`
	The left and right braces enclose a group of commands to be applied as a set to the range specified by *addr1* and *addr2*. The first command in the set can be on the line following the left brace as illustrated in this table, or it can be on the same line with the brace. The right brace must be on a line by itself. Groups can be nested within other groups.

Label	
`:`*label*	Marks a place in the stream of editing commands to be used as a destination of a branch command. The label is a string of up to 8 bytes. Each label in the editing stream must be unique. For a related discussion, see the description of the `t` command in the `sed`(1) reference page.
Branch	
`b`*label*	Branches to the point in the editing script indicated by *label* and continues processing the current input line with the commands following the label. If *label* is null, the `b` command bypasses the rest of the editing script, reads a new input line, and starts the editing script over from the beginning.
Conditional branch	
`t`*label*	If any successful substitutions were made on the current input line, branches to *label*; otherwise, the command does not branch. In either case, the command clears the flag that indicates a substitution was made. This flag is also cleared at the start of each new input line. If *label* is null and the branch is taken, the `t` command bypasses the rest of the editing script, reads a new input line, and starts the editing script over from the beginning.
Stop	
`[`*addr1*`]q`	Stops editing in an orderly fashion by writing the current line to the output, writing any appended or read text to the output, and then exiting.

3.4 String Replacement

The `s` command performs string replacement on the indicated lines in the input file. If the editor finds a string of characters in the input file that satisfies the RE *expr*, it replaces that string with the set of characters specified in *string*. The *string* argument is not an RE, and it is not scanned or otherwise interpreted except as follows:

- Any backslash characters (`\`) appearing in *string* must be escaped. See Table 3-3 for an explanation of how to handle slash characters (`/`) in *string*.
- The following two special symbols can be used in *string*:
  - Ampersand (`&`)

    This symbol in *string* is replaced by the exact string of characters in the input lines that matched *expr*. For example, apply the command `s/[Bb]oy/&s/` to the following line:

    ```
    The boy watched the game.
    ```

    This command tells `sed` to find either `Boy` or `boy` in the input line and copy whichever pattern it finds to the output with an appended "s". Since the command finds `boy`, it transfers that string to the output with the modification, and the result is as follows:

    ```
    The boys watched the game.
    ```

  - Repeat expression (`\`*n*)

    The number *n* is a single digit. This symbol in *string* is replaced by the string in the input line that matches the *n*th substring in *expr*. Substrings are delimited by backslash-parentheses sets `\(` and `\)`. For example, apply the command `s/\(stu\)\(dy\)/\1r\2/` to the following line:

    ```
    The study chair.
    ```

    This command tells `sed` to find `study` in the input line and copy that pattern to the output with an "r" inserted in the middle. The result is as follows:

    ```
    The sturdy chair.
    ```

You can modify the behavior of the `s` command with flags, as follows:

- Normally, only the first matching string in each line of the range is replaced. The `g` (global) flag causes `sed` to make the substitution for all matching strings anywhere on any line in the range. Note that the matching strings do not have to be identical; the RE *expr* is evaluated again for each potential match.

- The `p` (print) flag instructs `sed` to write the indicated lines explicitly after making any substitutions; this writing action is in addition to `sed`'s normal operation.
- The `w` *`file`* (write) flag instructs `sed` to write the indicated lines to the named file after making any substitutions. Include exactly one space between the `w` flag and the file name.

Any or all of these flags can be used with a given `s` command; in combinations, the `w` flag must be the last flag specified.

Creating Input Language Analyzers and Parsers 4

If a program needs to receive and process input, there must be a means of analyzing the input before it is processed. You can analyze input with one or more routines within the program, or with a separate program designed to filter the input before passing it to the main program. The complexity of the input interface depends on the complexity of the input; complicated input can require significant code to parse it (break it into pieces that are meaningful to the program). This chapter describes the following two tools that help develop input interfaces:

- The `lex` tool uses a set of rules to generate a program, called a **lexical analyzer**, that analyzes input and breaks it into categories, such as numbers, letters, or operators.
- The `yacc` tool uses a set of rules to generate a program, called a **parser**, that analyzes input using the categories identified by the lexical analyzer and determines what to do with the input. The `yacc` tool generates left-associative, left-recursive (LALR) parsers. For further information about LALR grammars, refer to a compiler book such as *Compilers: Principles, Techniques, and Tools*, by Alfred Aho, Ravi Sethi, and Jeffrey Ullman.[1]

To avoid confusion between the `lex` and `yacc` programs and the programs they generate, `lex` and `yacc` are referred to throughout this chapter as tools.

4.1 How the Lexical Analyzer Works

The lexical analyzer that `lex` generates is a **deterministic finite-state automaton**. This design provides for a limited number of states that the lexical analyzer can exist in, along with the rules that determine what state the lexical analyzer moves to upon reading and interpreting the next input character.

The compiled lexical analyzer performs the following functions:

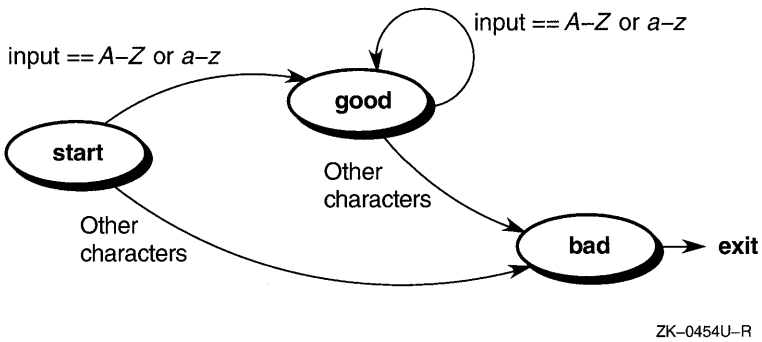
- Reads an input stream of characters.
- Copies the input stream to an output stream.

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*, Reading, MA, U.S.A.: Addison-Wesley Publishing Co., 1986.

- Breaks the input stream into smaller strings that match the regular expressions (REs) in the `lex` specification file.
- Executes an action for each RE that it recognizes. Actions are C language program fragments in the `lex` specification file. An action fragment does not have to be complete within itself; it can call subroutines or other actions.

Figure 4-1 illustrates a simple lexical analyzer that has three states: `start`, `good`, and `bad`. The program reads an input stream of characters. It begins in the `start` condition. When it receives the first character, the program compares the character with the rule. If the character is alphabetic (according to the rule), the program changes to the `good` state; if it is not alphabetic, the program changes to the `bad` state. The program stays in the `good` state until it finds a character that does not match its conditions, and then it moves to the `bad` state, which terminates the program.

Figure 4-1: Simple Finite State Model



The automaton allows the generated lexical analyzer to look ahead more than one or two characters in an input stream. For example, suppose the `lex` specification file defines a rule that looks for the string ''ab'' and another rule that looks for the string ''abcdefg''. If the lexical analyzer gets an input string of ''abcdefh'', it reads enough characters to attempt a match on ''abcdefg''. When the ''h'' disqualifies a match on ''abcdefg'', the analyzer returns to the rule that looks for ''ab''. The first two characters of the input match ''ab'', so the analyzer performs any action specified in that rule and then begins trying to find another match using the remaining input, ''cdefh''.

4.2 Writing a Lexical Analyzer Program with lex

The `lex` tool helps write a C language lexical analyzer program that can receive character stream input and translate that input into program actions.

To use `lex`, you must write a specification file that contains the following parts:

- Regular expressions (REs) – Character patterns that the generated lexical analyzer will recognize
- Action statements – C language program fragments that define how the generated lexical analyzer is to react to REs that it recognizes

The actual format and logic allowed in the specification file are discussed in Section 4.3.

The `lex` tool uses the information in the specification file to generate the lexical analyzer. The tool names the created analyzer program `yy.lex.c`. The `yy.lex.c` program contains a set of standard functions together with the analysis code that is generated from the specification file. The analysis code is contained in the `yylex` function. Lexical analyzers created by `lex` recognize simple grammar structures and REs. You can compile a simple `lex` analyzer program with the following command:

```
% cc -ll yy.lex.c
```

The `–ll` option tells the compiler to use the `lex` function library. This command yields an executable lexical analyzer. If your program uses complex grammar rules, or if it uses no grammar rules, you should create a parser (by combining the `lex` and `yacc` tools) to ensure proper handling of the input. (See Section 4.6.)

The `yy.lex.c` output file can be moved to any other system having a C compiler that supports the `lex` library functions.

4.3 The lex Specification File

The format of the `lex` specification file is as follows:

[{*definitions*}]
%%
[{*rules*}]
[%%
{*user subroutines*}]

Except for the first pair of percent signs (`%%`), which mark the beginning of the rules, all parts of the specification file are optional, even the rules themselves. The minimum `lex` specification file contains no definitions, no rules, and no user subroutines:

```
%%
```

Without a specified action for a pattern match, the lexical analyzer copies the input pattern to the output without changing it. Therefore, this minimum specification file produces a lexical analyzer that copies all input to the output unchanged.

4.3.1 Defining Substitution Strings

You can define string macros before the first pair of percent signs in the `lex` specification file. The `lex` tool expands these macros when it generates the lexical analyzer. Any line in this section that begins in column 1 and that does not lie between `%{` and `%` delimiters defines a `lex` substitution string. Substitution string definitions have the following general format:

name translation

The *name* and *translation* elements are separated by a least one blank or tab, and *name* begins with a letter. When `lex` finds the string *name* enclosed in braces (`{ }`) in the rules part of the specification file, it changes *name* to the string defined in *translation* and deletes the braces.

For example, to define the names `D` and `E`, place the following definitions before the first `%%` delimiter in the specification file:

```
D               [0-9]
E               [DEde][-+]{D}+
```

These definitions can be used in the rules section to make identification of integers and real numbers more compact:

```
{D}+            printf("integer");
{D}+"."{D}*({E})?|
{D}*"."{D}+({E})?|
{D}+{E}         printf("real");
```

You can also include the following items in the definitions section:

- Character set table (described in Section 4.3.3)
- List of start conditions (described in Section 4.3.6)
- Changes to size of arrays to accommodate larger source programs

4.3.2 Rules

The rules section of the specification file contains control decisions that define the lexical analyzer that `lex` generates. The rules are in the form of a two-column table. The left column of the table contains REs; the right column of the table contains actions, one for each RE. Actions are C language program fragments that can be as simple as a semicolon (the null statement) or as complex as needed. The lexical analyzer that `lex` creates

contains both the REs and the actions; when it finds a match for one of the REs, it executes the corresponding action.

For example, to create a lexical analyzer to look for the string ''integer'' and print a message when the string is found, define the following rule:

```
integer         printf ("found keyword integer");
```

This example uses the C language library function `printf` to print a message string. The first blank or tab character in the rule indicates the end of the RE. When you use only one statement in an action, put the statement on the same line and to the right of the RE (`integer` in this example). When you use more than one statement, or if the statement takes more than one line, enclose the action in braces, as in a C language program. For example:

```
integer         {
          printf ("found keyword integer");
          hits++;
          }
```

A lexical analyzer that changes some words in a file from British spellings to the American spellings would have a specification file that contains rules such as the following:

```
colour          printf("color");

mechanise       printf("mechanize");

petrol          printf("gas");
```

This specification file is not complete, however, because it changes the word ''petroleum'' to ''gaseum''.

4.3.2.1 Regular Expressions

Methods used to specify REs in a `lex` specification file are similar to those used in `sed` or `ed`. Chapter 1 contains a thorough exposition of REs. REs consist of text characters and operators. A text character is any character that is not an operator or a special character as described in Section 4.3.2.1.2. Table 4-1 lists the operators that `lex` recognizes. Note that some of these operators are unique to `lex`.

Table 4-1: Regular Expression Operators for lex

Operator	Name	Description
	Period (dot)	Matches any single character except the newline character.

Table 4-1: (continued)

Operator	Name	Description
*	Asterisk	Matches any number of occurrences of the preceding object, including none.
?	Question mark	Matches zero or one occurrence of the preceding object. For example, `ab?c` matches "ac" or "abc" but not "abbc".
+	Plus sign	Matches one or more occurrences of the preceding object.
\{*expr*\}	Braces	When enclosing numbers, matches a more restricted number of instances of the preceding object. When braces enclose a name, the name represents a string defined earlier in the specification file. For example, `{digit}` looks for a defined string named `digit` and inserts that string at the point in the expression where `{digit}` occurs.
[*chars*]	Brackets	Matches a single instance of any one of the characters within the brackets. Ranges of characters can be abbreviated by using a dash; for example, `[0-9a-z]` matches any single digit or lowercase letter. Note that if the program is moved to a system that uses a different set of character codes (for example, EBCDIC instead of ASCII), the range can be a different set of characters. Ranges of characters are referred to as classes.
^	Circumflex	When used at the beginning of an RE, matches the beginning of a line. When used as the first character inside brackets, excludes the bracketed characters from being matched. Otherwise, has no special properties.
$	Dollar sign	When used at the end of an RE, matches the end of a line. Otherwise, has no special properties.
char	Backslash	Escapes the next character to permit matching on explicit instances of characters that are normally RE operators. For example, `abc\+\+` matches "abc++".

Table 4-1: (continued)

Operator	Name	Description
`(expr)`	Parentheses	Encloses, or frames, an RE, allowing operators that act on the preceding object to treat the entire framed RE as an object. For example, `(ab)?(cd)` matches `cd` or `abcd`.
`expr|expr ...`	Vertical bar	Separates multiple REs; matches any of the bar-separated REs.
`" "`	Quotation marks	Encloses literal strings to interpret as text characters. For example, `"$"` prevents `lex` from interpreting the dollar sign as an operator. You can use quotation marks for only part of a string; for example, both `"abc++"` and `abc"++"` match the literal string "abc++".
a/b	Slash	Enables a match on the first expression (`a`) only if the second expression (`b`) follows it immediately. For example, `ab/cd` matches "ab" if, and only if, "cd" immediately follows the "ab".
`<X>`	Angle brackets	Encloses a start condition. Executes the associated action only if the lexical analyzer is in the indicated start condition `<x>`. If the condition of being at the beginning of a line is start condition `ONE`, then the circumflex (`^`) operator would be the same as the expression `<ONE>`.

4.3.2.1.1 Including Blanks in an Expression – Normally, white space (blanks or tabs) delimits the end of an RE and the start of its associated action. However, you can enclose blanks or tab characters in quotation marks (`" "`) to include them in an expression. Use quotation marks around all blanks in REs that are not already within sets of brackets (`[ ]`).

4.3.2.1.2 Other Special Characters – The `lex` tool recognizes several of the normal C language special characters. These character sequences are as follows:

`\n`	Newline character – Do not use the actual newline character in an expression

`\t`	Tab
`\b`	Backspace
`\\`	Backslash

When you use these special characters in an expression, you do not need to enclose them in quotation marks. Every character, except these special characters and the previously described operator symbols, is always a text character.

4.3.2.2 Matching Rules

When more than one RE in the rules section of a specification file can match the current input, the lexical analyzer chooses which rule to apply using the following criteria:

1. The longest matching string of characters
2. Among rules that match the same number of characters, the rule that occurs first

For example, consider the following rules:

```
integer         printf("found int keyword");
[a-z]+          printf("found identifier");
```

If the rules are given in this order and ''integers'' is the input word, the analyzer calls the input an identifier because `[a-z]+` matches all eight characters of the word while `integer` matches only seven. However, if the input is ''integer'', both rules match. In this case, `lex` selects the keyword rule because it occurs first. A shorter input, such as ''int'', does not match the expression `integer`, so `lex` selects the identifier rule.

4.3.2.2.1 Using Wildcard Characters to Match a String – Because the lexical analyzer chooses the longest match first, you must be careful not to use an RE that is too powerful for your intended purpose. For example, a period followed by an asterisk and enclosed in apostrophes (`'.*'`) might seem like a good way to recognize any string enclosed in apostrophes. However, the analyzer reads far ahead, looking for a distant apostrophe to complete the longest possible match. Consider the following text:

```
'first' quoted string here, 'second' here
```

Given this input, the analyzer will match on the following string:

```
'first' quoted string here, 'second'
```

Because the period operator does not match a newline character, errors of this type are usually not far reaching. Expressions like `.*` stop on the current line. Do not try to defeat this action with expressions like the following:

```
[.\n]+
```

Given this expression, the lexical analyzer tries to read the entire input file, and an internal buffer overflow occurs.

The following rule finds the smaller quoted strings ''first'' and ''second'' from the preceding text example:

```
'[^'\n]*'"
```

This rule stops after matching ''first'' because it looks for an apostrophe followed by any number of characters *except* another apostrophe or a newline character, then followed by a second apostrophe. The analyzer then begins again to search for an appropriate expression, and it will find ''second'' as it should. Note that this expression also matches an empty quoted string (`''`).

4.3.2.2.2 Finding Strings Within Strings – Normally, the lexical analyzer program partitions the input stream. It does not search for all possible matches of each expression. Each character is accounted for exactly once. For example, to count occurrences of both ''she'' and ''he'' in an input text, consider the following rules:

```
she         s++;
he          h++;
\n          ;
.           ;
```

The last two rules ignore everything other than the two strings of interest. However, because ''she'' includes ''he'', the analyzer does not recognize the instances of ''he'' that are included within ''she''.

A special action, `REJECT`, is provided to override this behavior. This directive tells the analyzer to execute the rule that contains it and then, before executing the next rule, restore the position of the input pointer to where it was before the first rule was executed. For example, to count the instances of ''he'' that are included within ''she'', use the following rules:

```
she         {s++; REJECT;}
he          {h++; REJECT;}
\n          ;
.           ;
```

After counting an occurrence of ''she'', the analyzer rejects the input stream and then counts the included occurrence of ''he''. In this example, ''she'' includes ''he'' but the reverse is not true, and you can omit the `REJECT` action on ''he''. In other cases, such as when a wildcard RE is being matched, determining which input characters are in both classes can be difficult.

In general, REJECT is useful whenever the purpose is not to partition the input stream but rather to detect all examples of some items in the input where the instances of these items can overlap or include each other.

4.3.2.3 Actions

When the lexical analyzer matches one of the REs in the rules section of the specification file, it executes the action that corresponds to the RE. Without rules to match all strings in the input stream, the lexical analyzer copies the input to standard output. Therefore, do not create a rule that only copies the input to the output. Use this default output to find conditions not covered by the rules.

When you use a `lex`-generated analyzer to process input for a parser that `yacc` produces, provide rules to match all input strings. Those rules must generate output that `yacc` can interpret. For information on using `lex` with `yacc`, see Section 4.5.

4.3.2.3.1 Null Action – To ignore the input associated with an RE, use a semicolon (`;`), the C language null statement, as an action. For example:

```
[ \t\n]          ;
```

This rule ignores the three spacing characters (blank, tab, and newline character).

4.3.2.3.2 Using the Same Action for Multiple Expressions – To use the same action for several different expressions, create a series of rules (one for each expression except the last) whose actions consist of only a vertical bar character (`|`). For the last expression, specify the action as you would normally specify it. The vertical bar character indicates that the action for the rule containing it is the same as the action for the next rule. For example, to ignore blank, tab, and newline characters (shown in Section 4.3.2.3.1), you could use the following set of rules:

```
" "      |
"\t"     |
"\n"     ;
```

The quotation marks around the special character sequences (`\n` and `\t`) in this example are not mandatory.

4.3.2.3.3 Printing a Matched String – To find out what text matched an RE in the rules section of the specification file, include a C language `printf` function as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts that matched string in an external character array, called `yytext`. To print the matched string, use a rule like the following:

```
[a-z]+          printf("%s", yytext);
```

Printing the output in this way is common. You can define an expression like this `printf` statement as a macro in the definitions section of the specification file. If this action is defined as `ECHO`, then the rules section entry looks like the following:

```
[a-z]+              ECHO;
```

See Section 4.3.1 for information on defining macros.

4.3.2.3.4 Finding the Length of a Matched String – To find the number of characters that the lexical analyzer matched for a particular RE, use the external variable `yyleng`. For example, the following rule counts both the number of words and the number of characters in words in the input:

```
[a-zA-Z]+       {words++; chars += yyleng;}
```

This action totals the number of characters in the words matched and assigns that value to the `chars` variable.

The following expression finds the last character in the string matched:

```
yytext[yyleng-1]
```

4.3.2.3.5 Getting More Input – The lexical analyzer can run out of input before it completely matches an RE in a rules file. In this case, include a call to the `lex` function `yymore` in the action for that rule. Normally, the next string from the input stream overwrites the current entry in `yytext`. The `yymore` action appends the next string from the input stream to the end of the current entry in `yytext`. For example, consider a language that includes the following syntax:

- A string is any set of characters between quotation marks (" ").
- A backslash (\ ) escapes the next character to make that character part of the string. For example, the combination of a backslash and a quotation mark (\") indicates that the quotation mark is part of the string instead of being the closing delimiter for the string.

The following rule processes these lexical characteristics:

```
\"[^"]* {
        if (yytext[yyleng-1] == '\\')
          yymore();
        else
          ... normal user processing
      }
```

When this lexical analyzer receives a string such as "abc\"def" (with the quotation marks exactly as shown), it first matches the first five characters, "abc\. The backslash causes a call to `yymore` to add the next part of the

string, "def, to the end. The part of the action code labeled ''normal user processing'' must process the quotation mark that ends the string.

4.3.2.3.6 Returning Characters to the Input – In some cases the lexical analyzer does not need all of the characters that are matched by the currently successful RE; or it might need to return matched characters to the input stream to be checked again for another match. To return characters to the input stream, use the `yyless(n)` call, where *n* is the number of characters of the current string that you want to keep. Characters beyond the *n*th character in the stream are returned to the input stream. This function provides the same type of look-ahead that the slash operator (`/`) uses, but `yyless` allows more control over the look-ahead. Using `yyless(0)` is equivalent to using `REJECT`.

Use the `yyless` function to process text more than once. For example, a C language expression such as `x=-a` is ambiguous. It could mean `x = -a`, or it could be an obsolete representation of `x -= a`, which is evaluated as `x = x - a`. To treat this ambiguous expression as `x = -a` and print a warning message, use a rule such as the following:

```
=-[a-zA-Z]     {
          printf("Operator (=-) ambiguous\n");
          yyless(yyleng-1);
          ... action for =-...
          }
```

4.3.3 Using or Overriding Standard Input/Output Routines

The `lex` program provides a set of built-in input/output (I/O) routines for the lexical analyzer to use. You can include calls to the following routines in the C code fragments in your specification file:

- `input` – Returns the next input character
- `output(c)` – Writes the character *c* on the output
- `unput(c)` – Pushes the character *c* back onto the input stream to be read later by input

These routines are provided as macro definitions. You can override them by writing your own code for routines of the same names in the user subroutines section.

These routines define the relationship between external files and internal characters. If you change them, change them all in the same way. They should follow these rules:

- All routines must use the same character set.

- The input routine must return a value of 0 to indicate end-of-file.

If you write your own code, you must undefine these macros in the definitions section of the specification file before the code for your own definitions:

```
%{
#undef    input
#undef    unput
#undef    output
}%
```

Caution

Changing the relationship of `unput` to `input` will cause the look-ahead functions not to work.

When you are using a `lex`-generated lexical analyzer as a simple transformer/recognizer for piping from standard input to standard output, you can avoid writing the ''framework'' by using the `lex` library (`libl.a`). This library contains the `main` routine, which calls the `yylex` function for you. The standard `lex` library allows the lexical analyzer to back up a maximum of 100 characters.

If you need to be able to read an input file containing the null character (000_8), you must create a different version of the `input` routine. The standard version of `input` returns a value of 0 when reading a null, and the analyzer interprets this value as indicating the end of the file.

The lexical analyzers that `lex` generates process character I/O through the `input`, `output`, and `unput` routines. Therefore, to return values in `yytext`, the analyzer uses the character representation that these routines use. Internally, however, each character is represented with a small integer. With the standard library, this integer is the value of the bit pattern that the computer uses to represent the character. Normally, the letter ''a'' is represented in the same form as the character constant `a`. If you change this interpretation with different I/O routines, you must include a translation table in the definitions section of the specification file. The translation table begins and ends with lines that contain only the `%T` keyword, and it contains lines of the following form:

{*integer***}** **{***character string***}**

The following example shows table entries that associate the letter ''A'' and the digit ''0'' (zero) with their standard values:

```
%T
{65} {A}
{48} {0}
%T
```

4.3.4 End-of-File Processing

When the lexical analyzer reaches the end of a file, it calls a library routine named `yywrap`. This routine returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up (operations associated with the end of processing). However, if the analyzer receives input from more than one source, you must change the `yywrap` function. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates that the program should continue processing. Note that multiple files specified on the command line are treated as a single input file for the purpose of end-of-file handling.

You can also include code to print summary reports and tables in a special version of `yywrap`. The `yywrap` function is the only way to force `yylex` to recognize the end of the input.

4.3.5 Passing Code to the Generated Program

You can define variables in either the definitions section or the rules section of the specification file. When you process a specification file, `lex` changes statements in the file into a lexical analyzer. Any line in the specification file that `lex` cannot interpret is passed unchanged to the lexical analyzer. The following four types of entries can be passed to the lexical analyzer in this manner:

- Lines beginning with a blank or tab that are not a part of a `lex` rule are copied into the lexical analyzer. If this entry occurs before the first pair of percent signs (`%%`) in the specification file, the entry is external to any function in the code. If the entry occurs after the first `%%`, it must be a C language program fragment that defines a variable. You must define these statements before the first `lex` rule in the specification file.
- Lines beginning with a blank or tab that are program comments are included as comments in the generated lexical analyzer. The comments must be in the C language format for comments.
- Any lines that lie between lines containing only `%{` and `%` are copied to the lexical analyzer. The symbols `%{` and `%}` are not copied. Use this format to enter preprocessor statements that must begin in column 1, or to copy lines that do not look like program statements.
- Any lines occurring after the third `%%` delimiter are copied to the lexical analyzer without format restrictions.

4.3.6 Start Conditions

Any rule can be associated with a start condition; the lexical analyzer recognizes that rule only when the analyzer is in that start condition. You can change the current start condition at any time.

You define start conditions in the definitions section of the specification file by using a line with the following format:

% **Start** *name1*[*name2* ...]

The *name1* and *name2* symbols represent conditions. There is no limit to the number of conditions, and they can appear in any order. You can abbreviate `Start` to either `S` or `s`. Start-condition names cannot be reserved words in C, nor can they be declared as the names of variables, fields, and so on.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in angle brackets (<>) at the beginning of the rule. The following format defines a rule with a start condition:

<*name1*[**,***name2* ...] **>** *expression*

The lexical analyzer recognizes *expression* only when the analyzer is in the condition corresponding to one of the names. To put `lex` in a particular start condition, execute the following action statement (in the action part of a rule):

BEGIN*name***;**

This statement changes the start condition to *name*. To resume the normal state, use the following action:

```
BEGIN 0;
```

As shown in the preceding syntax diagram, a rule can be active in several start conditions. For example:

```
<start1,start2,start3>  [0-9]+  printf("integer");
```

This rule prints ''integer'' only if it finds an integer while in one of the three named start conditions. Any rule that does not begin with a start condition is always active.

4.4 Generating a Lexical Analyzer

Generating a `lex`-based lexical analyzer program is a two-step process, as follows:

1. Run `lex` to change the specification file into a C language program. The resulting program is in a file named `lex.yy.c`.

2. Process `lex.yy.c` with the `cc -ll` command to compile the program and link it with a library of `lex` subroutines. The resulting executable program is named `a.out`.

For example, if the `lex` specification file is called `lextest`, enter the following commands:

```
% lex lextest
% cc lex.yy.c -ll
```

Although the default `lex` I/O routines use the C language standard library, the lexical analyzers that `lex` generates do not require them. You can include different copies of the `input`, `output`, and `unput` routines to avoid using those in the library. (See Section 4.3.3.)

Table 4-2 describes the options for the `lex` command.

Table 4-2: Options for the lex Command

Option	Description
`-n`	Suppresses the statistics summary that is produced by default when you set your own table sizes for the finite state machine. See the `lex`(1) reference page for information about specifying the state machine.
`-t`	Writes the generated lexical analyzer code to standard output instead of to the `lex.yy.c` file.
`-v`	Provides a one-line summary of the general finite state machine statistics.

Note that because `lex` uses fixed names for intermediate and output files, you can have only one `lex`-generated program in a given directory unless you use the `–t` option to specify an alternative file name.

4.5 Using lex with yacc

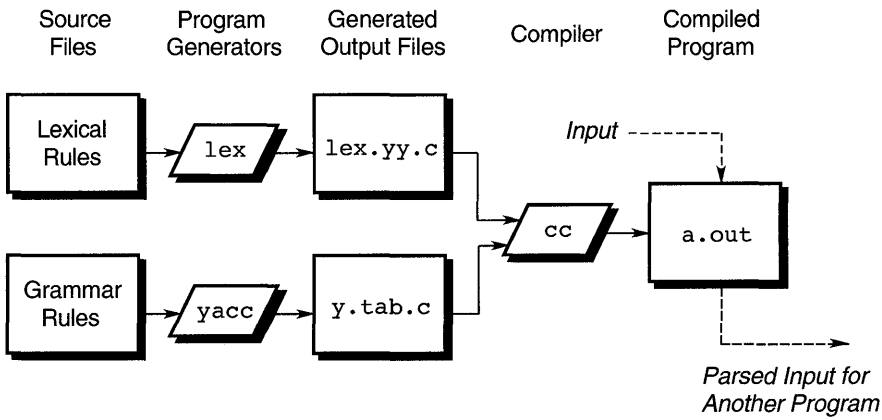
When used alone, the `lex` tool creates a lexical analyzer that recognizes simple one-word input or receives statistical input. You can also use `lex` with a parser generator, such as `yacc`. The `yacc` tool generates a program, called a parser, that analyzes the construction of multiple-word input. This parser program operates well with lexical analyzers that `lex` generates; these lexical analyzers recognize only REs and format them into character packages called **tokens**.

A token is the smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax. A token can be any string

of characters; it can be part or all of a word or series of words. The yacc tool produces parsers that recognize many types of grammar with no regard to context. These parsers need a preprocessor, such as a lex-generated lexical analyzer, to recognize input tokens.

When a lex-generated lexical analyzer is used as the preprocessor for a yacc-generated parser, the lexical analyzer partitions the input stream. The parser assigns structure to the resulting pieces. Figure 4-2 shows how lex and yacc generate programs and how the programs work together. You can also use other programs along with those generated by lex or yacc.

Figure 4-2: Producing an Input Parser with lex and yacc



The parser program requires that its preprocessor (the lexical analysis function) be named yylex. This is the name lex gives to the analysis code in a lexical analyzer it generates. If a lexical analyzer is used by itself, the default main program in the lex library calls the yylex routine, but if a yacc-generated parser is loaded and its main program is used, the parser calls yylex. In this case, each lex rule should end with the following line, where the appropriate token value is returned:

```
return(token);
```

To find the names for tokens that yacc uses, compile the lexical analyzer (the lex output file) as part of the parser (the yacc output file) by placing the following line in the last section of the yacc grammar file:

```
#include lex.yy.c
```

Alternatively, you can include the `yacc` output (the `y.tab.h` file) into your `lex` program specification file, and use the token names that `y.tab.h` defines. For example, if the grammar file is named `good` and the specification file is named `better`, the following command sequence creates the final program:

```
% yacc good
% lex better
% cc y.tab.c -ly -ll
```

The `yacc` library (`-ly` in the preceding example) should be loaded before the `lex` library to get a `main` program that invokes the `yacc` parser. You can generate `lex` and `yacc` programs in either order.

4.6 Creating a Parser with yacc

To generate a parser with `yacc`, you must write a grammar file that describes the input data stream and what the parser is to do with the data. The grammar file includes rules describing the input structure, code to be invoked when these rules are recognized, and a routine to do the basic input.

The `yacc` tool uses the information in the grammar file to generate `yyparse`, a program that controls the input process. This is the parser that calls the `yylex` input routine (the lexical analyzer) to pick up tokens from the input stream. The parser organizes these tokens according to the structure rules in the grammar file. The structure rules are called grammar rules. When the parser recognizes a grammar rule, it executes the user code (action) supplied for that rule. Actions return values and use the values returned by other actions.

In addition to the specifications that `yacc` recognizes and uses, the grammar file can also contain the following functions:

- `main` – A C language function that contains, as a minimum, a call to the `yyparse` function, which `yacc` generates. A limited form of this function is in the `yacc` library.
- `yyerror` – A C language function to handle errors that can occur during parser operation. A limited form of this function is in the `yacc` library.
- `yylex` – A C language function to perform lexical analysis on the input stream and pass tokens (with values, if required) to the parser.

  The function must return an integer that represents the kind of token that was read. The integer is called the token number. In addition, if a value is associated with the token, the lexical analyzer must assign that value to the external variable `yylval`. See Section 4.7.1.3 for more information on token numbers.

To build a lexical analyzer that works well with the parser that yacc generates, use the lex tool (see Section 4.3).

The yacc tool processes a grammar file to generate a file of C language functions and data, named y.tab.c. When compiled using the cc command, these functions form a combined function named yyparse. This yyparse function calls yylex, the lexical analyzer, to get input tokens. The analyzer continues providing input until the parser detects an error or the analyzer returns an endmarker token to indicate the end of the operation. If an error occurs and yyparse cannot recover, yyparse returns a value of 1 to the main function. If it finds the endmarker token, yyparse returns a value of 0 to main.

Use the C programming language to write the action code and other subroutines. The yacc program uses many of the C language syntax conventions for the grammar file.

4.6.1 The main and yyerror Functions

You must provide function routines named main and yyerror in the grammar file. To ease the initial effort of using yacc, the yacc library provides simple versions of the main and yyerror routines. You can include these routines by using the -ly option to the loader or the cc command. The source code for the main library function is as follows:

```
main()
{
     yyparse();
}
```

The source code for the yyerror library function follows:

```
#include <stdio.h>

void yyerror(s)
        char *s;
{
        fprintf( stderr, " %s\n" ,s);
}
```

The argument to yyerror is a string containing an error message, usually the string syntax error.

These are very limited programs. You should provide more sophistication in these routines, such as keeping track of the input line number and printing it along with the message when a syntax error is detected. You can also use the value of the external integer variable yychar. This variable contains the look-ahead token number at the time the error was detected.

4.6.2 The yylex Function

The `yylex` program input routine that you supply must be able to do the following:

- Read the input stream
- Recognize basic patterns in the input stream
- Pass the patterns to `yyparse` along with tokens that identify them

A token is a symbol or name that tells `yyparse` which pattern is being sent to it by the input routine. A symbol can be in one of the following two classes:

- Terminal symbols – Values returned by `yylex` to represent the primitive building blocks, or ''atoms,'' of the grammar.
- Nonterminal symbols – The composite symbols, or ''molecules,'' that are used by the `yacc` grammar to describe more complex orderings or aggregations of the terminal symbols.

For example, if the lexical analyzer recognizes any numbers, names, and operators, these elements are taken to be terminal symbols. Nonterminal symbols that the `yacc` grammar recognizes are elements like `EXPR`, `TERM`, and `FACTOR`. Suppose the input routine separates an input stream into the tokens of `WORD`, `NUMBER`, and `PUNCTUATION`. Consider the input sentence ''I have 9 turkeys.'' The analyzer could pass the following strings and tokens to the parser:

String	Token
I	`WORD`
have	`WORD`
9	`NUMBER`
turkeys	`WORD`
.	`PUNCTUATION`

The `yyparse` function must contain definitions for the tokens that the input routine passes to it. The `yacc` command's `–d` option causes the program to generate a list of tokens in a file named `y.tab.h`. This list is a set of `#define` statements that allow `yylex` to use the same tokens as the parser.

To avoid conflict with the parser, do not use names that begin with the letters `yy`. You can use `lex` to generate the input routine, or you can write it in the C language. See Section 4.3 for information about using `lex`.

4.7 The Grammar File

A `yacc` grammar file consists of the following three sections:

- Declarations
- Rules
- Programs

Two percent signs (`%%`) that appear together separate the sections of the grammar file. To make the file easier to read, put the percent signs on a line by themselves. The format of a grammar file is:

[*declarations*]
%%
rules
[**%%**
programs]

Except for the first pair of percent signs (`%%`), which mark the beginning of the rules, and the rules themselves, all parts of the grammar file are optional. The minimum `yacc` gramar file contains no definitions and no programs, as follows:

```
%%
rules
```

Except within names or reserved symbols, the `yacc` program ignores blanks, tabs, and newline characters in the grammar file. You can use these characters to make the grammar file easier to read. Do not use blanks, tabs, or newline characters in names or reserved symbols.

4.7.1 Declarations

The declarations section of the `yacc` grammar file contains the following elements:

- Declarations for any variables or constants used in other parts of the grammar file
- `#include` statements to call in other files as part of this file (used for library header files)
- Statements that define processing conditions for the generated parser

Declarations for variables or constants conform to the syntax of the C programming language, as follows:

type-specifier declarator;

In this syntax, *`type-specifier`* is a data type keyword and *`declarator`* is the name of the variable or constant. Names can be any

length and can consist of letters, dots, underscores, and digits. A name cannot begin with a digit. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule can represent tokens or nonterminal symbols.

If you do not declare a name in the declarations section, you can use that name only as a nonterminal symbol. Define each nonterminal symbol by using it as the left side of at least one rule in the rules section. The `#include` statements are identical to C language syntax and perform the same function.

The `yacc` tool has a set of keywords, listed in Table 4-3, that define processing conditions for the generated parser. Each of the keywords begins with a percent sign (`%`) and is followed by a list of tokens.

Table 4-3: Processing-Condition Definition Keywords in yacc

Keyword	Description
`%left`	Identifies tokens that are left-associative with other tokens.
`%nonassoc`	Identifies tokens that are not associative with other tokens.
`%right`	Identifies tokens that are right-associative with other tokens.
`%start`	Identifies a name for the start symbol.
`%token`	Identifies the token names that `yacc` accepts. Declare all token names in the declarations section.

All tokens listed on the same line have the same precedence level and associativity; lines appear in the file in order of increasing precedence or binding strength. For example:

```
%left     '+'  '-'
%left     '*'  '/'
```

This example describes the precedence and associativity of the four arithmetic operators. The addition (+) and subtraction (–) operators are left-associative and have lower precedence than the multiplication (*) and division (/) operators, which are also left-associative.

4.7.1.1 Defining Global Variables

You can define global variables to be used by some or all parser actions, as well as by the lexical analyzer, by enclosing the declarations for those variables in matched pairs of symbols consisting of a percent sign and a brace (`%{` and `%}`). For example, to make the `var` variable available to all parts of the complete program, place the following entry in the declarations section of the grammar file:

```
%{ int var = 0; %}
```

4.7.1.2 Start Symbols

The parser recognizes a special symbol called the start symbol. The start symbol is the name assigned to the grammar rule that describes the most general structure of the language to be parsed. Because it is the most general structure, it is the structure where the parser starts in its top-down analysis of the input stream. You declare the start symbol in the declarations section by using the `%start` keyword. If you do not declare a start symbol, the parser uses the name of the first grammar rule in the file.

For example, in parsing a C language procedure, the following is the most general structure for the parser to recognize:

```
main()
{
   code_segment}
```

The start symbol should point to the rule that describes this structure. All remaining rules in the file describe ways to identify lower-level structures within the procedure.

4.7.1.3 Token Numbers

Token numbers are nonnegative integers that represent the names of tokens. Since the lexical analyzer passes the token number to the parser instead of the actual token name, the programs must assign the same numbers to the tokens.

You can assign numbers to the tokens used in the `yacc` grammar file. If you do not assign numbers to the tokens, `yacc` assigns numbers using the following rules:

- A literal character is assigned the numeric value of the character in the ASCII character set.
- Other names are assigned token numbers starting at 257.

Note

Do not assign a token number of 0 (zero). This number is assigned to the `endmarker` token. You cannot redefine it.

To assign a number to a token (including literals) in the declarations section of the grammar file, put a nonzero positive integer immediately after the token name in the `%token` line. This integer is the token number of the name or literal. Each number must be unique. Any lexical analyzer used with `yacc` must return either 0 (zero) or a negative value for a token when the end of the input is reached.

4.7.2 Grammar Rules

The rules section of the `yacc` grammar file contains one or more grammar rules. Each rule describes a structure and gives it a name. A grammar rule has the following format:

nonterminal-name : *BODY* ;

In this syntax, *BODY* is a sequence of zero or more names and literals. The colon and the semicolon are required `yacc` punctuation.

If there are several grammar rules with the same nonterminal name, use the vertical bar (|) to avoid rewriting the left side. In addition, use the semicolon (;) only at the end of all rules joined by vertical bars. The two following sets of grammar rules are equivalent:

Set 1

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

Set 2

```
A  :  B  C  D
   |  E  F
   |  G
   ;
```

4.7.2.1 The Null String

To indicate a nonterminal symbol that matches the null string, use a semicolon by itself in the body of the rule, as follows:

```
nullstr  :  ;
```

4.7.2.2 End-of-Input Marker

When the lexical analyzer reaches the end of the input stream, it sends a special token, called `endmarker`, to the parser. This token signals the end of the input and has a token value of 0. When the parser receives an `endmarker` token, it checks to see that it has assigned all of the input to defined grammar rules and that the processed input forms a complete unit (as defined in the `yacc` grammar file). If the input is a complete unit, the parser stops. If the input is not a complete unit, the parser signals an error and stops.

The lexical analyzer must send the `endmarker` token at the correct time, such as the end of a file, or the end of a record.

4.7.2.3 Actions in yacc Parsers

With each grammar rule, you can specify actions to be performed each time the parser recognizes the rule in the input stream. Actions return values and obtain the values returned by previous actions. The lexical analyzer can also return values for tokens.

An action is a C language statement that does input and output, calls subprograms, and alters external vectors and variables. You specify an action in the grammar file with one or more statements enclosed in braces (`{ }`). For example, the following are grammar rules with actions:

```
A   :   '('B')'
    {
      hello(1, "abc" );
    };
XXX   :   YYY   ZZZ
      {
      printf("a message\n");
      flag = 25;
      }
```

An action can receive values generated by other actions by using numbered `yacc` parameter keywords (`$1`, `$2`, and so on). These keywords refer to the values returned by the components of the right side of a rule, reading from left to right. For example:

```
A   :   B   C   D   ;
```

When this code is executed, `$1` has the value returned by the rule that recognized `B`, `$2` the value returned by the rule that recognized `C`, and `$3` the value returned by the rule that recognized `D`.

To return a value, the action sets the pseudovariable `$$` to some value. For example, the following action returns a value of 1:

```
{ $$ = 1;}
```

By default, the value of a rule is the value of the first element in it (`$1`). Therefore, you do not need to provide actions for rules that have the following form:

```
A : B ;
```

To get control of the parsing process before a rule is completed, write an action in the middle of a rule. If this rule returns a value through the `$`*n* parameters, actions that come after it can use that value. The action can use values returned by actions that come before it. Therefore, the following rule sets `x` to 1 and `y` to the value returned by `C`:

```
A  :  B
          {
             $$ =1;
          }
      C
          {
             x = $2;
             y = $3;
          }
    ;
```

Internally, `yacc` creates a new nonterminal symbol name for the action that occurs in the middle, and it creates a new rule matching this name to the null string. Therefore, `yacc` treats the preceding program as if it were written in the following form, where `$ACT` is an empty action:

```
$ACT :     /* null string */
     {
      $$ = 1;
     }
     ;
A    :     B   $ACT   C
     {
       x = $2;
       y = $3;
     }
     ;
```

4.7.3 Programs

The programs section of the `yacc` grammar file contains C language functions that can be used by the actions in the rules section. In addition, if you write a lexical analyzer (`yylex`, the input routine to the parser), include it in the programs section.

4.7.4 Guidelines for Using Grammar Files

This section describes some general guidelines for using `yacc` grammar files. It provides information on the following:

- Using comments
- Using literal strings
- Formatting grammar files
- Using recursion
- Correcting errors

4.7.4.1 Using Comments

Comments in the grammar file explain what the program is doing. You can put comments anywhere in the grammar file that you can put a name. However, to make the file easier to read, put the comments on lines by themselves at the beginning of functional blocks of rules. Comments in a `yacc` grammar file have exactly the same form as comments in a C language program; that is, they begin with a slash and an asterisk (`/*`) and end with an asterisk and a slash (`*/`). For example:

```
/* This is a comment on a line by itself. */
```

4.7.4.2 Using Literal Strings

A literal string is one or more characters enclosed in apostrophes, or single quotation marks (`' '`). As in the C language, the backslash (`\`) is an escape character within literals, and all the C language special-character sequences are recognized, as follows:

`\n`	Newline character
`\r`	Return
`\'`	Apostrophe, or single quote
`\\`	Backslash
`\t`	Tab
`\b`	Backspace
`\f`	Form feed
`\`*nnn*	The value *nnn* in octal

Never use `\0` or `0` (the null character) in grammar rules.

4.7.4.3 Guidelines for Formatting the Grammar File

The following guidelines will help make the `yacc` grammar file more readable:

- Use uppercase letters for token names and lowercase letters for nonterminal symbol names.
- Put grammar rules and actions on separate lines to allow for changing either one without changing the other.
- Put all rules with the same left side together. Enter the left side once and use vertical bars (`|`) to begin the rest of the rules for that left side.
- For each set of rules with the same left side, enter the semicolon (`;`) once on a line by itself following the last rule for that left side. You can then add new rules easily.

- Indent rule bodies by two tab stops and action bodies by three tab stops.

4.7.4.4 Using Recursion in a Grammar File

Recursion is the process of using a function to define itself. In language definitions, these rules normally take the following form:

```
rule      :    end case
          |    rule, end case
```

The simplest case of `rule` is the end case, but `rule` can also be made up of more than one occurrence of *`end case`*. The entry in the second line that uses `rule` in the definition of `rule` is the instance of recursion. Given this rule, the parser cycles through the input until the stream is reduced to the final end case.

The `yacc` tool supports left-recursive grammar, not right-recursive. When you use recursion in a rule, always put the call to the name of the rule as the leftmost entry in the rule (as it is in the preceding example). If the call to the name of the rule occurs later in the line, as in the following example, the parser can run out of internal stack space and crash:

```
rule      :    end case
          |    end case, rule
```

4.7.4.5 Errors in the Grammar File

The `yacc` tool cannot produce a parser for all sets of grammar specifications. If the grammar rules contradict themselves or require matching techniques different from those that `yacc` has, `yacc` will not produce a parser. In most cases, `yacc` provides messages to indicate the errors. To correct these errors, redesign the rules in the grammar file or provide a lexical analyzer to recognize the patterns that `yacc` cannot handle.

4.7.5 Error Handling by the Parser

When the parser reads an input stream, that input stream can fail to match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error-handling routine in the grammar file, the parser can allow for entering the data again, skipping over the bad data, or for a cleanup and recovery action. When the parser finds an error, for example, it might need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating any further output.

When an error occurs, the parser stops unless you provide error-handling routines. To continue processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard

some of the tokens following the error, and try to restart the parser at that point in the input stream.

The `yacc` tool has a special token name, `error`, to use for error handling. Put this token in your grammar file at places where an input error might occur so that you can provide a recovery routine. If an input error occurs in a position protected by the `error` token, the parser executes the action for the `error` token rather than the normal action.

To prevent a single error from producing many error messages, the parser remains in an error state until it successfully processes three tokens following an error. If another error occurs while the parser is in the error state, the parser discards the input token and does not produce a message. You can also specify a point at which the parser should resume processing by providing an argument to the `error` action. For example:

```
stat  :  error ';'
```

This rule tells the parser that, when there is an error, it should skip over the token and all following tokens until it finds the next semicolon. All tokens after the error and before the next semicolon are discarded. When the parser finds the semicolon, it reduces this rule and performs any cleanup action associated with it.

4.7.5.1 Providing for Error Correcting

You can allow the person entering the input stream in an interactive environment to correct any input errors by reentering a line in the data stream. For example:

```
input   :    error '\n'
        {
          printf(" Reenter last line: " );
        }
        input
       {
         $$ = $4;
       }
       ;
```

However, in this example the parser stays in the error state for three input tokens following the error. If the corrected line contains an error in the first three tokens, the parser deletes the tokens and does not display a message. To allow for this condition, use the `yyerrok;` statement. When the parser

encounters the `yyerrok;` statement, it leaves the error state and begins normal processing. The error recovery example then becomes the following:

```
input   :    error      '\n'
        {
          yyerrok;
          printf( "Reenter last line: " );
        }
        input
       {
         $$ = $4
       }
       ;
```

4.7.5.2 Clearing the Look-Ahead Token

The look-ahead token is the next token to be examined by the parser. When an error occurs, the look-ahead token becomes the token at which the error was detected. However, if the error recovery action includes code to find the correct place to start processing again, that code must also change the look-ahead token. To clear the look-ahead token, include the `yyclearin;` statement in the error recovery action.

4.8 Parser Operation

The `yacc` program turns the grammar file into a C language program that, when compiled and executed, parses the input according to the grammar rules.

The parser is a finite state machine with a stack. The parser can read and remember the next input token (the look-ahead token). The current state is always the state that is on the top of the stack. The states of the finite state machine are represented by small integers. Initially, the machine is in state 0 (zero), the stack contains only 0 (zero), and no look-ahead token has been read.

The machine can perform one of the following four actions:

`shift` *n*
: The parser pushes the current state onto the stack, makes *n* the current state, and clears the look-ahead token.

`reduce` *r*
: The *r* argument is a rule number. When the parser finds a token sequence matching rule number `r` in the input stream, the parser replaces that sequence with the rule number in the output stream.

`accept`
: The parser has looked at all input, matched it to the grammar specification, and recognized the input as satisfying the highest level structure (defined by the start symbol). This action appears only when the look-ahead token is the end marker and indicates that the parser has successfully done its job.

`error` The parser cannot continue processing the input stream and still successfully match it with any rule defined in the grammar specification. The input tokens it looked at, together with the look-ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing.

The parser performs the following actions during one process step:

1. Based on its current state, the parser decides whether it needs a look-ahead token to decide the action to take. If it needs one and does not have one, it calls `yylex` to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This can result in states being pushed onto the stack or popped off the stack and in the look-ahead token being processed or left alone.

4.8.1 The shift Action

The `shift` action is the most common action the parser takes. Whenever the parser does a `shift`, there is always a look-ahead token. Consider the following parser action rule:

```
IF shift 34
```

When the parser is in the state that contains this rule and the look-ahead token is `IF`, the parser performs the following steps:

1. Pushes the current state down on the stack
2. Makes state 34 the current state (puts it on the top of the stack)
3. Clears the look-ahead token

4.8.2 The reduce Action

The `reduce` action prevents the stack from growing too large. The parser uses reducing actions after it has matched the right side of a rule with the input stream and is ready to replace the tokens in the input stream with the left side of the rule. The parser might have to use the look-ahead token to decide if the pattern is a complete match.

Reducing actions are associated with individual grammar rules. Because grammar rules also have small integer numbers, you can easily confuse the meanings of the numbers in the `shift` and `reduce` actions. For example,

the first of the two following actions refers to grammar rule 18; the second refers to machine state 34:

```
reduce 18
IF shift 34
```

For example, consider reducing the following rule:

```
A  :  x y z ;
```

The parser pops off the top three states from the stack. The number of states popped equals the number of symbols on the right side of the rule. These states are the ones put on the stack while recognizing `x`, `y`, and `z`. After popping these states, the parser uncovers the state the parser was in before beginning to process the rule (the state that needed to recognize rule `A` to satisfy its rule). Using this uncovered state and the symbol on the left side of the rule, the parser performs a `goto` action, which is similar to a `shift` of `A`. A new state is obtained and pushed onto the stack, and parsing continues.

The `goto` action is different from an ordinary `shift` of a token. The look-ahead token is cleared by a `shift` but is not affected by a `goto`. When the three states are popped in this example, the uncovered state contains an entry such as the following:

```
A  goto 20
```

This entry causes state 20 to be pushed onto the stack and become the current state.

The `reduce` action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the parser executes the code that you included in the rule before adjusting the stack. 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 executing the code that you provide, the parser performs the reduction. When the parser performs the `goto` action, it copies the external variable `yylval` onto the value stack. The `yacc` variables whose names begin with a dollar sign (`$`) refer to the value stack.

4.8.3 Ambiguous Rules and Parser Conflicts

A set of grammar rules is ambiguous if any possible input string can be structured in two or more different ways. For example:

```
expr : expr '-' expr
```

This rule forms an arithmetic expression by putting two other expressions together with a minus sign between them, but this grammar rule does not specify how to structure all complex inputs. For example:

```
expr - expr - expr
```

Using the preceding rule, a program could structure this input as either left-associative or right-associative:

```
(  expr - expr ) - expr
```

or

```
expr - ( expr - expr )
```

These two forms produce different results when evaulated.

When the parser tries to handle an ambiguous rule, it can become confused over which of its four actions to perform when processing the input. The following two types of conflicts develop:

`Shift/reduce` conflict — A rule can be evaluated correctly using either a `shift` action or a `reduce` action, with different results.

`Reduce/reduce` conflict — A rule can be evaluated correctly using one of two different `reduce` actions, producing two different actions.

A `shift/shift` conflict is not possible.

These conflicts result when a rule is not as complete as it could be. For example, consider the following input and the preceding ambiguous rule:

```
a - b - c
```

After reading the first three parts of the input, the parser has the following:

```
a - b
```

This input matches the right side of the grammar rule. The parser can reduce the input by applying this rule. After applying the rule, the input becomes the following:

expr

This is the left side of the rule. The parser then reads the final part of the input, as follows:

```
- c
```

The parser now has the following:

expr **- c**

Reducing this input produces a left-associative interpretation.

However, the parser can also look ahead in the input stream. If, after receiving the first three parts of the input, it continues reading the input stream until it has the next two parts, it then has the following input:

```
a - b - c
```

Applying the rule to the rightmost three parts reduces `b - c` to *expr*. The parser then has the following:

```
a - expr
```

Reducing the expression once more produces a right-associative interpretation.

Therefore, at the point where the parser has read the first three parts, it can take one of two legal actions: a `shift` or a `reduce`. If the parser has no rule by which to decide between the actions, a `shift/reduce` conflict results.

A similar situation occurs if the parser can choose between two valid `reduce` actions. That situation is called a `reduce/reduce` conflict.

When `shift/reduce` or `reduce/reduce` conflicts occur, `yacc` produces a parser by selecting a valid step wherever it has a choice. If you do not provide a rule to make the choice, `yacc` uses the following rules:

- In a `shift/reduce` conflict, shift.
- In a `reduce/reduce` conflict, reduce by the grammar rule that can be applied at the earliest point in the input stream.

Using actions within rules can cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, using the preceding rules leads to an incorrect parser. For this reason, `yacc` reports the number of `shift/reduce` and `reduce/reduce` conflicts that it has resolved by applying its rules.

4.9 Turning on Debug Mode

For normal operation, the external integer variable `yydebug` is set to 0. However, if you set it to any nonzero value, the parser generates a running description of the input tokens that it receives and the actions that it takes for each token. You can set the `yydebug` variable in one of the following two ways:

- Use the `yydebug` function by including the following C language statement in the declarations section of the `yacc` grammar file:

```
yydebug = 1;
```

- Use a debugger to execute the final parser, and set the `yydebug` variable on or off using debugger commands. For further details about using debuggers, such as `dbx`, see the reference pages for the various debuggers.

4.10 Creating a Simple Calculator Program

This section describes the programs for a `lex`-generated lexical analyzer and a `yacc`-generated parser. These programs together create a simple desk calculator program that performs addition, subtraction, multiplication, and division operations. The calculator program also allows you to assign values to variables (each designated by a single lowercase letter) and then use the variables in calculations. The files that contain the programs are as follows:

- `calc.l` – The `lex` specification file that defines the lexical analysis rules
- `calc.y` – The `yacc` grammar file that defines the parsing rules, and calls the `yylex` function created by `lex` to provide input

By convention, `lex` and `yacc` programs use the letters `.l` and `.y` respectively as file name suffixes. Example 4-1 and Example 4-2 contain the program fragments exactly as they should be entered. The processing instructions in this section assume that the files are in your current directory.

Perform the following steps, in the order shown, to create the calculator program using `lex` and `yacc`:

1. Process the `yacc` grammar file by using the following command. The `–d` option tells `yacc` to create a file that defines the tokens it uses in addition to the C language source code.

   ```
   % yacc -d calc.y
   ```

   This command creates the following files:

   - `y.tab.c` – The C language source file that `yacc` created for the parser
   - `y.tab.h` – A header file containing `define` statements for the tokens used by the parser

2. Process the `lex` specification file by using the following command:

   ```
   % lex calc.l
   ```

   This command creates the `lex.yy.c` file, containing the C language source file that `lex` created for the lexical analyzer.

3. Compile and link the two C language source files by using the following command:

   ```
   % cc -o calc y.tab.c lex.yy.c
   ```

4. Use the `ls` command to verify that the following files were created:

   - `y.tab.o` – The object file for `y.tab.c`
   - `lex.yy.o` – The object file for `lex.yy.c`

- `calc` – The executable program file

You can run the program by entering the `calc` command. You can then enter numbers and operators in algebraic fashion. After you press Return, the program displays the result of the operation. You can assign a value to a variable as follows:

```
m=4
```

You can use variables in calculations as follows:

```
m+5
9
```

4.10.1 The Parser Source Code

Example 4-1 shows the contents of the `calc.y` file. This file has entries in all three of the sections of a `yacc` grammar file: declarations, rules, and programs. The grammar defined by this file supports the usual algebraic hierarchy of operator precedence.

Descriptions of the various elements of the file and their functions follow the example.

Example 4-1: Parser Source Code for a Calculator

```
%{
#include <stdio.h>     1

int regs[26];     2
int base;

%}

%start list     3

%token DIGIT LETTER     4

%left '|'     5
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*supplies precedence for unary minus */

%%                    /* beginning of rules section */

list:                          /*empty */
          | list stat'\n'
          | list error'\n'
          {
            yyerrok;
          }
          ;
```

Example 4-1: (continued)

```
stat:    expr
         {
           printf("%d\n",$1);
         }
         |
         LETTER '=' expr
         {
           regs[$1] = $3;
         }
         ;
expr:    '(' expr ')'
         {
           $$ = $2;
         }
         |
         expr '*' expr
         {
           $$ = $1 * $3;
         }
         |
         expr '/' expr
         {
           $$ = $1 / $3;
         }
         |
         expr '%' expr
         {
           $$ = $1 % $3;
         }
         |
         expr '+' expr
         {
           $$ = $1 + $3;
         }
         |
         expr '-' expr
         {
           $$ = $1 - $3;
         }
         |
         expr '&' expr
         {
           $$ = $1 & $3;
         }
         |
         expr '|' expr
         {
           $$ = $1 | $3;
         }
         |
         '-' expr %prec UMINUS
         {
           $$ = -$2;
```

Example 4-1: (continued)

```
          }
          |
          LETTER
          {
            $$ = regs[$1];
          }
          |
          number
          ;
number:   DIGIT
          {
            $$ = $1;
            base = ($1==0) ? 8:10;
          }
          |
          number DIGIT
          {
            $$ = base * $1 + $2;
          }
          ;
%%
main()
{
 return(yyparse());
}

yyerror(s)
char *s;
{
  fprintf(stderr," %s\n",s);
}

yywrap()
{
  return(1);
}
```

The declarations section contains entries that perform the following functions:

1 Include standard I/O header file

2 Define global variables

3 Define the rule list as the place to start processing

4 Define the tokens used by the parser

5 Define the operators and their precedence

The rules section defines the rules that parse the input stream.

The programs section contains the following routines. Because these routines are included in this file, you do not need to use the `yacc` library when processing this file.

- `main( )` – The required main program that calls `yyparse( )` to start the program
- `yyerror(s)` – The error-handling routine, which prints a syntax error message
- `yywrap( )` – The wrap-up routine that returns a value of 1 when the end of input occurs

4.10.2 The Lexical Analyzer Source Code

Example 4-2 shows the contents of the `calc.l` file. This file contains `#include` statements for standard input and output and for the `y.tab.h` file, which is generated by `yacc` before you run `lex` on `calc.l`. The `y.tab.h` file contains definitions for the tokens that the parser program uses. In addition, `calc.l` contains the rules to generate the tokens from the input stream.

Example 4-2: Lexical Analyzer Source Code for a Calculator

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "         ;
[a-z]       {
              c = yytext[0];
              yylval = c - 'a';
              return(LETTER);
            }
[0-9]       {
              c = yytext[0];
              yylval = c - '0';
              return(DIGIT);
            }
[^a-z0-9\b]     {
                  c = yytext[0];
                  return(c);
                }
```

Using m4 Macros in Your Programs 5

This chapter describes the m4 macro preprocessor, a front-end filter that allows you to define macros by placing m4 macro definitions at the beginning of your source files. The m4 preprocessor can be used with either program source files or document source files.

5.1 Using Macros

Macros ease your programming or writing tasks by allowing you to substitute a simple word or two for a great amount of material. Macro calls in a source file have the following form:

name[(*arg1*[,*arg2*...])]

For example, suppose you have a C program in which you want to print the same message at several points. You could code a series of `printf` statements like the following:

```
printf("\nThese %d files are in %s:\n\n",cnt,dir);
```

As your program evolves, you decide to change the wording; but you have to edit each instance of the message. Defining a macro like the following will save you a great deal of work:

```
define(filmsg,'printf("\nThese%d files are in %s:\n\n",$1,$2)')
```

Then, everywhere you want to output this message, you use the macro this way:

```
filmsg(cnt,dir);
```

With this implementation, you need only edit the message in one place.

A **macro definition** consists of a symbolic name (called a **token**) and the character string that is to replace it. A token is any string of alphanumeric characters (letters, numbers, and underscores) beginning with a letter or an underscore and delimited by nonalphanumeric characterss (punctuation or white space). For example, `N12` and `N` are both tokens but `A+B` is not a token. When you process your file through m4, each occurrence of a recognized macro is replaced by its definition. In addition to replacing symbolic names with text, m4 can also perform the following operations:

- Arithmetic calculation

- File manipulation
- Conditional macro expansion
- String and substring functions
- System command execution

The m4 program reads each token in the file and determines if the token is a macro name. Macro names that are embedded in other tokens are not recognized; for example, m4 does not interpret N12 as containing an occurrence of the token N. If the token is a macro name, m4 replaces it with its defining text and pushes the resulting string back onto the input to be rescanned. Macro expansion is thus recursive; macro definitions can include nested occurrences of other macros to any depth of nesting. You can call macros with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The m4 preprocessor is a standard UNIX filter. It accepts input from standard input or from a list of input files and writes its output to standard output. The following lines illustrate correct m4 usage:

```
% grep -v '#include' file1 file2 | m4 > outfile
% m4 file1 file2 | cc
```

The m4 program processes each argument in order. If there are no arguments, or if an argument is a minus sign (–), m4 reads standard input as its input file.

5.2 Defining Macros

You create a macro definition with the `define` command, one of about 20 built-in macros provided by m4. For example:

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

The open parenthesis must follow the word `define` with no intervening space.

Given this macro definition, the token N will be replaced by 100 wherever it appears in the file being processed. The defining text can be any text, except that if the text contains parentheses, the number of open (left) parentheses must match the number of close (right) parentheses unless you protect an unmatched parenthesis by quoting it. See Section 5.2.1 for an explanation of quoting.

Built-in and user-defined macros work the same way except that some of the built-in macros change the state of the process. Refer to Section 5.3 for a list of the built-in macros.

You can define macros in terms of other macros. For example:

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

This example defines both M and N to be 100. If you later change the definition of N and assign it a new value, M retains the value of 100, not the new value you give N. The value of M does not track that of N because the m4 preprocessor expands macro names into their defining text as soon as possible. The overall result, as far as M is concerned, is the same as using the following input in the first place:

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

If you want the value of M to track the value of N, you can reverse the order of the definitions, as follows:

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

Now M is defined to be the string N. When the value of M is requested later, the M is replaced by N, which is then rescanned and replaced by whatever value N has at that time.

Macro definitions made with the `define` command do not delete characters following the close parenthesis. For example:

```
Now is the time for all good persons.
define(N,100)
Testing N definition.
```

This example produces the following result:

```
Now is the time for all good persons.

Testing 100 definition.
```

The blank line results from the presence of a newline character at the end of the line containing the `define` macro. The built-in `dnl` macro deletes all characters that follow it, up to and including the next newline character. Use this macro to delete empty lines. For example:

```
Now is the time for all good persons.
define(N,100)dnl
Testing N definition.
```

This example produces the following result:

```
Now is the time for all good persons.
Testing 100 definition.
```

5.2.1 Using the Quote Characters

To delay the expansion of a `define` macro's arguments, enclose them in a matched pair of quote characters. The default quote characters are left and right single quotation marks (‘ and ’), but you can use the built-in

`changequote` macro to specify different characters. (See Section 5.3.) Any text surrounded by quote characters is not expanded immediately, but the quote characters are removed. The value of a quoted string is the string with the quote characters removed. Consider the following example:

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

The quote characters around the `N` are removed as the argument is being collected. The result of using quote characters is to define `M` as the string `N`, not `100`. This example makes the value of `M` track that of `N`, and it is thus another way to accomplish the effect of the following definitions, shown in Section 5.2:

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

The general rule is that `m4` always strips off one level of quote characters whenever it evaluates something. This is true even outside macros. For example, to make the word "define" appear in the output, enter the word in quote characters, as follows:

```
'define' = 1
```

Because of the way `m4` handles quoted strings, you must be careful about nesting macros. For example:

```
define(dog,canid)
define(cat,animal chased by 'dog')
define(mouse,animal chased by cat)
```

When the definition of `cat` is processed, `dog` is not replaced with `canid` because it is quoted. But when `mouse` is processed, the definition of `cat` (`animal chased by dog`) is used; this time, `dog` is not quoted, and the definition of `mouse` becomes `animal chased by animal chased by canid`.

When you redefine an existing macro, you must quote the first argument (the macro name), as follows:

```
define(N,100)
   •
   •
   •
define('N',200)
```

Without the quote characters, the second `define` macro sees `N`, recognizes it, and substitutes its value, producing the following result:

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

The `m4` program ignores this statement because it can only define names, not numbers.

5.2.2 Macro Arguments

The simplest form of macro processing is replacing one string with another (fixed) string as illustrated in the previous sections. However, macros can also have arguments, so that you can use a given macro in different places with different results. To indicate where an argument is to be used within the replacement text for a macro (the second argument of its definition), use the symbol `$n` to indicate the *n*th argument. For example, the symbol `$1` refers to the first argument of a macro. When the macro is used, `m4` replaces the symbol with the value of the indicated argument. For example:

```
define(bump,$1=$1+1)
   •
   •
   •
bump(x);
```

In this example, `m4` will replace the `bump(x)` statement with `x=x+1`.

A macro can have as many arguments as needed. However, you can access only nine arguments by using the `$n` symbols (`$1` through `$9`). To access arguments past the ninth argument, use the `shift` macro, which drops the first argument and reassigns the remaining arguments to the `$n` symbols (second argument to `$1`, third to `$2`, and so on). Using the `shift` macro more than once allows access to all arguments used with the macro.

The symbol `$0` returns the name of the macro. Arguments that are not supplied are replaced by null strings, so that you can define a macro that concatenates its arguments as follows:

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

This example replaces the `cat(x,y,z)` statement with `xyz`. Arguments `$4` through `$9` in this example are null because corresponding arguments were not provided.

When scanning a macro, the `m4` program discards leading unquoted blanks, tabs, or newline characters in arguments, but keeps all other white space. For example:

```
define(a,     "$1 $2$3")
   •
   •
   •
a(b,
c,
d)
```

This example expands the a macro to be "b cd". In the define macro, however, newline characters are meaningful. For example:

```
define(a,$1
$2$3)
   •
   •
   •
a(b,c,d)
```

This latter example expands the a macro as follows:

```
b
cd
```

Macro arguments are separated by commas. Use parentheses to enclose arguments containing commas, so that the commas are not misinterpreted as ending the arguments containing them. For example, the following statement has only two arguments:

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

The first argument is a, and the second is (b,c). To use a single parenthesis in an argument, enclose it in quote characters:

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

In this example, b)c is the second argument.

5.3 Using Other m4 Macros

The m4 program provides a set of macros that are already defined (built-in macros). Table 5-1 lists all of these macros and describes them briefly. The following sections further explain many of the macros and how to use them.

Table 5-1: Built-In m4 Macros

Macro	Description
changecom(*l*,*r*)	Changes the left and right comment characters to the characters represented by *l* and *r*. The two characters must be different.
changequote(*l*,*r*)	Changes the left and right quote characters to the characters represented by *l* and *r*. The two characters must be different.
decr(*n*)	Returns the value of *n*–1.
define(*name*,*replacement*)	Defines a new macro, named name, with a value of *replacement*.
defn(*name*)	Returns the quoted definition of name.

Table 5-1: (continued)

Macro	Description
`divert(`*n*`)`	Changes the output stream to the temporary file number *n*.
`divnum`	Returns the number of the currently active temporary file.
`dnl`	Deletes text up to a newline character.
`dumpdef(`*'name'*`[,`*'name'*`...])`	Prints the names and current definitions of the named macros.
`errprint(`*str*`)`	Prints *str* to the standard error file.
`eval(`*expr*`)`	Evaluates *expr* as a 32-bit arithmetic expression.
`ifdef(`*'name',arg1,arg2*`)`	If macro *name* is defined, returns *arg1*; otherwise, returns *arg2*.
`ifelse(`*str1,str2,arg1,arg2*`)`	Compares the strings `str1` and `str2`. If they match, `ifelse` returns the value of `arg1`; otherwise, it returns the value of `arg2`.
`include(`*file*`)` `sinclude(`*file*`)`	Returns the contents of *file*. The `sinclude` macro does not report an error if it cannot access the file.
`incr(`*n*`)`	Returns the value of *n*+1.
`index(`*str1,str2*`)`	Returns the character position in string `str1` where `str2` starts, or –1 if `str1` does not contain `str2`.
`len(`*str*`)` `dlen(`*str*`)`	Returns the number of characters in `str`. The `dlen` macro operates on strings containing 2-byte representations of international characters.
`m4exit(`*code*`)`	Exits `m4` with a return code of *code*.
`m4wrap(`*name*`)`	Runs macro `name` before exiting, after completing all other processing.
`maketemp(`*str*`XXXXX`*str*`)`	Creates a unique file name by replacing the literal string `XXXXX` in the argument string with the current process ID.
`popdef(`*name*`)`	Replaces the current definition of `name` with the previous definition, saved with the `pushdef` macro.

Table 5-1: (continued)

Macro	Description
`pushdef(`*`name,replacement`*`)`	Saves the current definition of `name` and then defines `name` to be `replacement` in the same way as `define`.
`shift(`*`param_list`*`)`	Shifts the parameter list leftward one position, destroying the original first element of the list.
`substr(`*`string,pos,len`*`)`	Returns the substring of *`string`* that begins at character position *`pos`* and is *`len`* characters long.
`syscmd(`*`command`*`)`	Executes the specified system command with no return value.
`sysval`	Gets the return code from the last use of the `syscmd` macro.
`traceoff(`*`macro_list`*`)`	Turns off trace for any macro in the list. If `macro_list` is null, turns off all tracing.
`traceon(`*`name`*`)`	Turns on trace for the named macro. If `name` is null, turns trace on for all macros.
`translit(`*`string,set1,set2`*`)`	Replaces any characters from *`set1`* that appear in *`string`* with the corresponding characters from *`set2`*.
`undefine('`*`name`*`')`	Removes the definition of the named macro.
`undivert(`*`n`*`,`*`n`*`[,`*`n...`*`])`	Appends the contents of the indicated temporary files to the current temporary file.

5.3.1 Changing the Comment Characters

To include comments in your `m4` programs, delimit the comment lines with the comment characters. The default left comment character is the number sign (#); the default right comment character is the newline character. If these characters are not convenient, use the built-in `changecom` macro. For example:

```
changecom({,})
```

This example makes the left and right braces the new comment characters. To restore the original comment characters, use `changecom` as follows:

```
changecom(#,
)
```

Using `changecom` with no arguments disables commenting.

5.3.2 Changing the Quote Characters

The default quote characters are the left and right single quotation marks (`‘` and `’`). If these characters are not convenient, change the quote characters with the built-in `changequote` macro. For example:

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

This example makes the left and right brackets the new quote characters. To restore the original quote characters, use `changequote` without arguments, as follows:

```
changequote
```

5.3.3 Removing a Macro Definition

The `undefine` macro removes macro definitions. For example:

```
undefine('N')
```

This example removes the definition of `N`. Note that you must quote the name of the macro to be undefined. You can use `undefine` to remove built-in macros, but once you remove a built-in macro, you cannot recover that macro for later use.

5.3.4 Checking for a Defined Macro

The built-in `ifdef` macro determines if a macro is currently defined. The `ifdef` macro accepts three arguments. If the first argument is defined, the value of `ifdef` is the second argument. If the first argument is not defined, the value of `ifdef` is the third argument. If there is no third argument, the value of `ifdef` is null.

5.3.5 Using Integer Arithmetic

The `m4` program provides the following built-in functions for doing arithmetic on integers only:

`incr`	Increments its numeric argument by 1
`decr`	Decrements its numeric argument by 1
`eval`	Evaluates an arithmetic expression

For example, you can create a variable `N1` such that its value will always be one greater than `N`, as follows:

```
define(N,100)
define(N1,'incr(N)')
```

The `eval` function can evaluate expressions containing the following operators (listed in decreasing order of precedence):

- unary + (plus), unary – (minus)
- `**` or `^` (exponentiation)
- `*`, `/`, `%` (modulo)
- +, –
- ==, !=, <, <=, >, >=
- `!` (NOT)
- `&` or `&&` (logical AND)
- | or || (logical OR)

Use parentheses to group operations where needed. All operands of an expression must be numeric. The numeric value of a true relation such as `1>0` is 1, and false is 0 (zero). The precision in `eval` is 32 bits. For example, to define `M` as `2==N+1`, use `eval` as follows:

```
define(N,3)
define(M,'eval(2==N+1)')
```

Use quote characters around the text that defines a macro, unless the text is simple and contains no instances of macro names.

5.3.6 Manipulating Files

To merge a new file in the input, use the built-in `include` macro as follows:

```
include(myfile)
```

This example inserts the contents of `myfile` in place of the `include` command. As the included file is read, `m4` scans it for macros as if it were part of the primary input.

With the `include` macro, a fatal error occurs if the named file cannot be accessed. To avoid an error, use the alternative form, `sinclude` (silent include). The `sinclude` macro continues without error if the named file cannot be accessed.

5.3.7 Redirecting Output

You can redirect the output of m4 to temporary files during processing, and the collected material can be output upon command. The m4 program can maintain up to nine temporary files, numbered 1 through 9. To redirect output, use the divert macro as in the following example:

```
divert(4)
```

When this comand is encountered, m4 begins writing its output to the end of temporary file 4. The m4 program discards the output if you redirect the output to a temporary file other than 1 through 9; you can use this feature to make m4 omit a portion of the input file. Use divert(0) or divert with no argument to return the output to the standard output stream.

At the end of its processing, m4 writes all redirected output to the standard output stream, reading from the temporary files in numeric order and then destroying the temporary files. To retrieve the information from all temporary files in numeric order at any time before processing is completed, use the built-in undivert macro with no arguments. To retrieve selected temporary files in a specified order, use undivert with arguments. When using undivert, m4 discards the temporary files that are recovered and does *not* search the recovered information for macros.

Note that the value of undivert is not the diverted text.

The built-in divnum macro returns the number of the currently active temporary file. If you do not change the output file with the divert macro, m4 puts all output in temporary file 0 (zero).

5.3.8 Using System Programs in a Program

You can run any program in the operating system from a program by using the built-in syscmd macro. If the system command returns information, that information is the value of the syscmd macro; otherwise, the macro's value is null. For example:

```
syscmd(date)
```

5.3.9 Using Unique File Names

Use the built-in maketemp macro to make a unique file name from a program. If the literal string XXXXX is present in the macro's argument, m4 replaces the XXXXX with the process ID of the current process. For example:

```
maketemp(myfileXXXXX)
```

If the current process ID is 23498, this example returns `myfile23498`. You can use this string to name a temporary file.

5.3.10 Using Conditional Expressions

The built-in `ifelse` macro performs conditional testing. The simplest form is the following:

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

This example compares the two strings `a` and `b`. If they are identical, `ifelse` returns string `c`. If they are not identical, it returns string `d`. For example, you can define a macro called `compare` to compare two strings and return `yes` if they are the same or `no` if they are different, as follows:

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

The quote characters prevent the evaluation of `ifelse` from occurring too early. If the fourth argument is missing, it is treated as empty.

The `ifelse` macro can have any number of arguments, and it therefore provides a limited form of multiple path decision capability. For example:

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

This statement is logically the same as the following fragment:

```
if(a == b) x = c;
else if(d == e) x = f;
else  x = g;
return(x);
```

If the final argument is omitted, the result is null.

5.3.11 Manipulating Strings

The built-in `len` macro returns the byte length of the string that makes up its argument. For example, `len(abcdef)` is 6, and `len((a,b))` is 5.

The built-in `dlen` macro returns the length of the displayable characters in a string. In certain international usages, 2-byte codes are displayed as one character. Thus, if the string contains any 2-byte international character codes, the result of `dlen` will differ from the result of `len`.

The built-in `substr` macro returns the substring (beginning at the character position specified by the second argument) from a specified string (first argument). The third argument specifies the length in bytes of the returned substring. For example:

```
substr(Krazy Kat,6,5)
```

This example returns ''Kat'', which is the 3-character substring beginning at character position 6 of the string ''Krazy Kat''. The first character in the string is at position 0 (zero). If the third argument is omitted or if the string is not long enough to satisfy the third argument, as in this example, the rest of the string is returned.

The built-in `index` macro returns the byte position, or index, in a string (first argument) where a substring (second argument) begins. If the substring is not present, `index` returns –1. As with `substr`, the origin for strings is 0 (zero). For example:

```
index(Krazy Kat,Kat)
```

This example returns 6.

The built-in `translit` macro performs one-for-one character substitution, or transliteration. The first argument is a string to be processed. The second and third arguments are lists of characters. Each instance of a character from the second argument that is found in the string is replaced by the corresponding character from the third argument. For example:

```
translit(the quick brown fox jumps over the lazy dog,aeiou,AEIOU)
```

This example returns the following:

```
thE qUIck brOwn fOx jUmps OvEr thE lAzy dOg
```

If the third argument is shorter than the second argument, characters from the second argument that are not in the third argument are deleted. If the third argument is missing, all characters present in the second argument are deleted.

Note

The `substr`, `index`, and `translit` macros do not differentiate between 1- and 2-byte displayable characters and can return unexpected results in some international usages.

5.3.12 Printing

The built-in `errprint` macro writes its arguments to the standard error file. For example:

```
errprint ('error')
```

The built-in `dumpdef` macro dumps the current names and definitions of items named as arguments. Names must be quoted. If you supply no arguments, `dumpdef` prints all current names and definitions. The `dumpdef` macro writes to the standard error file.

Revision Control: Managing Source Files with RCS or SCCS 6

This chapter describes how to keep your program or documentation source files well organized by using a **version control** system. A version control system automates the storage, retrieval, logging, identification, and merging of document revisions. Version control is most useful for text that is revised frequently, such as programs, documentation, graphics, papers, and so on. The DEC OSF/1 operating system provides the following two version control systems with slightly different features:

- Revision Control System (RCS)
- Source Code Control System (SCCS)

Using RCS or SCCS allows you to keep your source files in a common library and maintain control over them. Both systems provide easy-to-use, command-line interfaces. Knowing the basic commands allows you to **check in** the source file to be modified into a **version control file** that contains all of the revisions of that source file. When you want to **check out** a version control file for editing, the system retrieves the revision or revisions you specify from the library and creates a **working file** for you to use.

Using more advanced interface commands allows you to do the following:

- Identify the current status of any file, including the name of the person editing it.
- Reconstruct earlier versions of your files. For each version, the system stores the changes made to produce that version, the name of the person making the changes, and the reasons for the changes.
- Prevent the problems that can occur when two people change a file at the same time without each other's knowledge.
- Maintain multiple branch versions of your files. Branched versions can be merged back into the original sequence if desired.
- Protect files from unauthorized modification.
- RCS also allows for release and configuration control. Revisions can be assigned symbolic names and marked according to the "state" of the file (for example, released, stable, experimental, and so on).

This chapter introduces basic version control concepts and describes how to use the RCS and SCCS commands and utilities. After the introduction, more

advanced uses of each system are described. Examples in this chapter describe a hypothetical kit for a product called "Orpheus Authoring Tools." The example kit is considered to be one of several Orpheus products. Because this particular kit is a document builder, the kit name is abbreviated as DCB and the main project directory is `dcb_tools`.

Depending on your development environment and unique revision control requirements, you can elect to use either RCS or SCCS as your version control system. Your choice will ultimately depend on the amount of security and versatility you require. Table 6-1 summarizes some of the more widely used features of each system.

Table 6-1: Features of RCS and SCCS

Feature	Comments
Stores and retrieves multiple revisions of text.	Both systems provide a simple way to store and retrieve all changes made to a file. In addition, RCS can retrieve revisions based on ranges of revision numbers, symbolic names, dates, authors, and states.
Maintains a complete history of changes.	Both systems log changes automatically. Besides the text of each revision, both systems store the author, date and time of the checkin, and a log message summarizing the changes.
Resolves access conflicts.	Both systems prevent two people from modifying a file without each other's knowledge.
Maintains tree of revisions.	Both systems can maintain separate lines of development for each file.
Merges revised files with conflict resolution.	Both systems provide a way to merge changes to a file from two separate lines of development. RCS also alerts the user if there are overlapping changes to the file versions.
Allows for release and configuration control. *(RCS only)*	RCS can assign symbolic names to revisions so that configurations of modules can be described simply and directly.
Automates identification of each revision.	Both systems use keywords to tag revisions of files with name, revision number, time, author, and so on.

6.1 Version Control Concepts

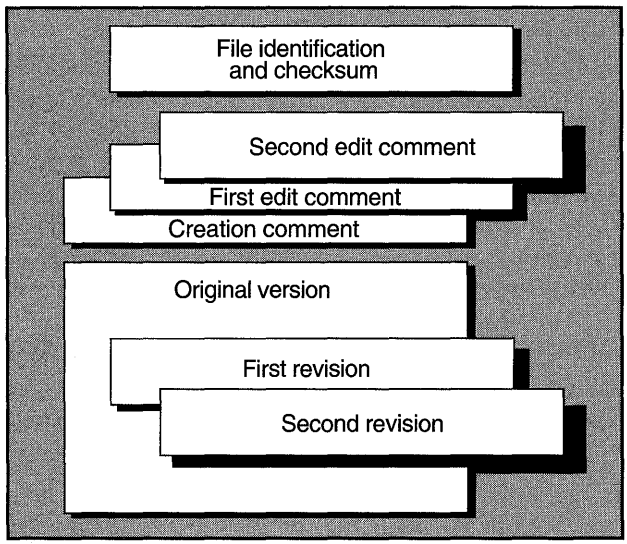
This section describes the contents of a version control file and the organization of a typical version control library. Key terms are introduced and explained.

RCS and SCCS store files in a reserved directory, called a **version control library**. The contents of each source file are stored as a single version control file (called an **RCS-file** in RCS or an **s-file** in SCCS). A version control file contains the original file (called a **g-file** in SCCS) together with all the changes, or **deltas**, that have been applied to it. Each delta is described by text telling who made the change and why. The change information itself is stored in the form of marked lines of text. Every line that is deleted or changed is marked as deleted but is not actually removed. New lines can be either edited versions of old lines or completely new material inserted at the appropriate places and marked. Your version control system can reconstruct any version of the file by applying all the deletions and additions for versions up to the desired version and by ignoring all later versions.

In RCS, RCS-files are identified by the suffix `,v` added to their names; for example, `attr,v` would be the RCS-file for the source file named `attr`.

In SCCS, s-files are identified by the prefix `s.` added to their names; for example, `s.attr` would be the s-file for the source file named `attr`. Figure 6-1 illustrates the contents of a typical version control file. RCS and SCCS files contain the same kinds of information, but their organization is different.

Figure 6-1: Contents of a Version Control File



ZK-0456U-R

A version identification number is applied to a particular revision of the version control file. In SCCS, this number is called an **SID**. The identification number for SCCS can contain up to four elements; RCS provides for additional elements. The first two elements are the release number and the level number within that release, and the third and fourth represent the same items of information (called the branch and the sequence) for a branched version of the file. (See Section 6.2.) Release identification numbers begin at 1. Level identification numbers begin at .1 and advance by .1, so that the first version of a file is 1.1, the second version is 1.2, and so on. Figure 6-4 (in Section 6.2) illustrates the numbering sequence for one file's deltas.

A version control library is a directory in which all the version control files for a given project are stored. When you retrieve a file from the library, both RCS and SCCS provide a **locking mechanism** that prevents two people from accessing the file at the same time. File locking will be discussed in more detail in the following sections.

Usually, but not always, the library is given the name `RCS` or `SCCS`, depending on the system you use. Figure 6-2 and Figure 6-3 illustrate how the DCB project's directory tree might appear with the RCS or SCCS library placed below the project's main directory.

Figure 6-2: A Typical RCS Library

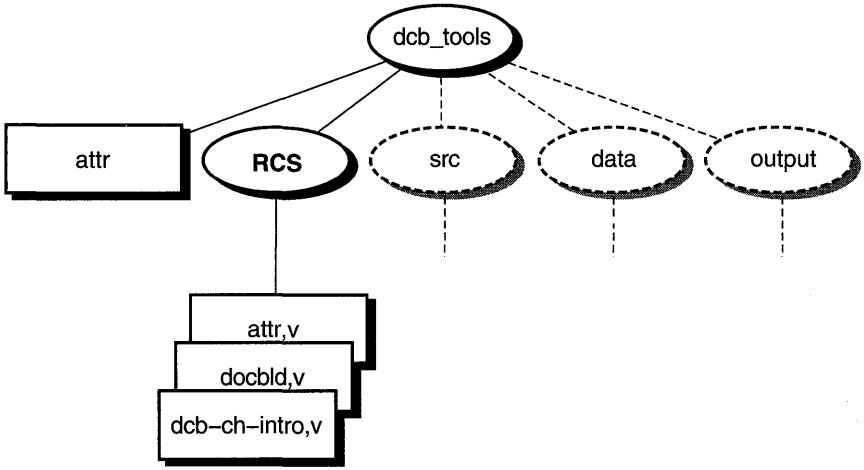


Figure 6-2 shows three RCS-files. When a file is checked out of the library for editing, RCS correlates all the deltas and delivers a copy of the specified version, as illustrated here with the `attr` file. RCS also edits the RCS-file to insert the name of the person checking out the file. This information is stored in the `$Locker$` keyword. See Section 6.4.2 for more information about using keywords in RCS.

RCS differs from SCCS in that file locking is enforced at checkin time. A file can be checked out by more than one person, but only the first person to check it out (the one holding the lock) can check it back in to the library. Even if a revision is locked, it can still be checked out for reading, compiling, and so on. Locking ensures that only one developer at a time can check in the next update of the file. In other words, locking prevents a checkin by anybody but the locker (the first person to check out the file).

If your RCS-file is private and you will be the only person making revisions to it, you can turn off the strict locking feature of RCS. When a file is checked back in, RCS removes the user's name from the `$Locker$` keyword. If strict locking is turned off, the owner of the file does not need to have a lock for checkin, but all others do. Use the following commands to turn strict locking off and on:

```
% rcs -U filename
```

and

```
% rcs -L filename
```

For more detailed information on file locking, refer to Section 6.4.3, Section 6.4.5, and the co(1) reference page.

Figure 6-3: A Typical SCCS Library

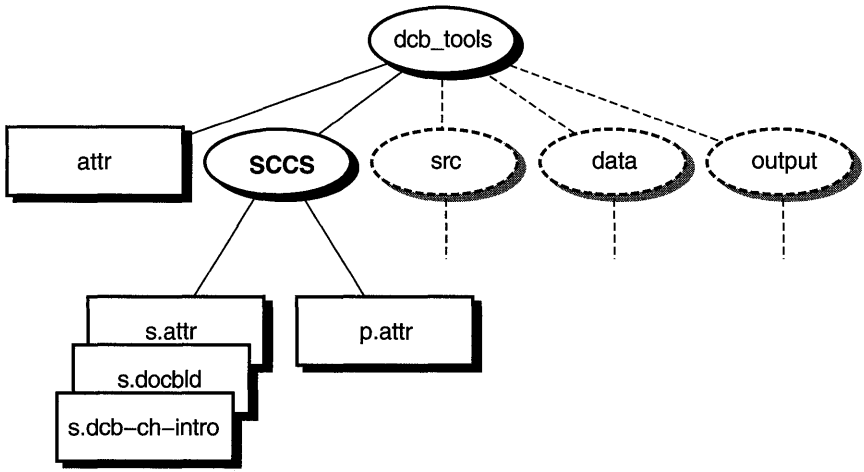


Figure 6-3 shows three s-files and one other file, named `p.attr`, in the SCCS library. When a file is checked out of the library for editing, SCCS correlates all the deltas and delivers a copy of the specified version, as illustrated here with the `attr` file. SCCS also creates a lock file, called a **p-file**. If another person tries to check out the same file for editing, SCCS reports that the file is being edited and refuses to give access to the second person. A p-file has the letter `p` added as a prefix to its name. When a file is checked back in to the library, SCCS removes the p-file.

6.2 Managing Multiple Versions of Files

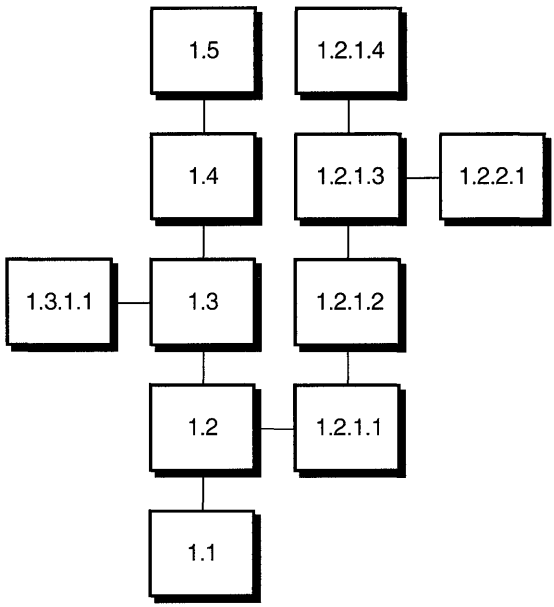
Normally, file versions progress in a straight line, with only one current version. In this case, file identification numbers contain two elements and progress by steps of .1, so that the first version number applied to a file is 1.1 and the eighth, for example, is 1.8.

Projects running in parallel to develop new versions of the same basic program can use the same version control file. As the different versions are put into the library, a tree develops. For example, suppose two teams begin

development on separate versions of a file or module, starting from the most recent revision.

As the two development streams continue, a complex tree of deltas can be created, as illustrated in Figure 6-4.

Figure 6-4: A Version Control File's Tree of Deltas



ZK-0458U-R

To get or edit a file from one of the branches, you must specify its branch number. Figure 6-4 shows a tree for a version control file that consists of a main trunk (contains revisions numbers 1.1, 1.2, 1.3, and so on) and branches. For the delta numbers shown, the first two elements reflect the version number from which it is branched, and the second two elements reflect the new element's version number.

As an example, suppose the two development teams are working with revision number 1.2 of a file. Both RCS and SCCS will allocate a number of 1.3 to the first team to access the file. For the second team, the version control system will create a delta numbered 1.2.1.1. Since this is the first delta along this 1.2 branch, the last two elements of this version number are shown as 1.1.

As the two versions are developed, they can themselves be branched from; for example, a programmer might branch a new file from revision number 1.2.1.3 after revision number 1.2.1.4 has been created.

For more information and specific examples on branching in RCS and SCCS, see Section 6.4.5 and Section 6.5.5.

6.3 Creating a Version Control Library

Once you have selected the version control system you want to use for your development project, you should create a directory in which you will place the RCS or SCCS files. Depending on the size and complexity of your development project, you might want to involve your system administrator, who can help you determine ownership and protection settings for the directory and source files.

When setting up your directory, you might want to assign ownership of the directory to the `rcs` or `sccs` user ID and set its permissions to prevent users other than `rcs` or `sccs` from writing to it. This method provides good security in that only RCS or SCCS can directly manipulate the files in the library. If you are going to use the `sccs` command, the library's directory should be named `SCCS`, as illustrated in Figure 6-3. If the library directory is not named `SCCS`, you must use the `–d` option with the `sccs` command to access files in the library. (See Table 6-8.) For RCS, the directory should be named `RCS`; otherwise, you must specify a complete path (absolute or relative) to the RCS-file.

6.4 Using RCS

The RCS system provides a set of UNIX commands that assist in the task of version control for your text files. It is designed for both production and development environments where flexibility and file access control are high priorities. In production environments, access controls can detect update conflicts and prevent overlapping changes. In fast-changing development environments, where such strong controls may not be appropriate, users can easily modify the controls to suit individual project needs.

The RCS system comprises a set of independent commands. Table 6-2 lists the RCS commands provided with DEC OSF/1. The sections following the table provide more information on some of these commands. Refer to the appropriate reference page for additional information on the available command options.

Table 6-2: Summary of RCS Command Functions

Command	Option	Description
`ci`		Checks in revisions. Stores the contents of a working file in the corresponding RCS-file as a new revision.

Table 6-2: (continued)

Command	Option	Description
	`-u` or `-l`	Using one of these options prevents a working file from being deleted at checkin time.
	`-r`	Assigns a revision number to the file being checked in.
	`-k`	Searches the checked-in file for identification markers, and assigns them to new revisions.
`co`		Checks out revisions. Retrieves revisions according to revision number, date, author, and state attributes. Always expands the identification markers (keywords).
	`-l`	Locks the revision during file checkout to prevent overlapping modifications if several people work on the same file.
`ident`		Extracts the identification markers from a file and prints them. The identification markers (keywords) are always expanded by `co`.
`rcs`		Changes RCS-file attributes. Changes (as an administrative operation) access lists, modifies file locking attributes, sets state attributes and symbolic revision numbers, changes the description, and deletes revisions. A revision can only be deleted if it is not the fork of a side branch.
	`-L`	Sets file locking to strict. This means that the owner of an RCS-file must lock the file at checkin. This default is determined by the system administrator.
	`-U`	Sets file locking to nonstrict. This means that the owner of the file does not need to lock the file at checkin. This default is determined by the system administrator.
`rcsclean`		Cleans your working directory. Removes working files that were checked out but never changed.
`rcsdiff`		Compares two revisions and prints out their differences, using the `diff` command. One of the revisions compared can be checked out. This command is useful for finding out about changes.
`rcsfreeze`		Freezes a configuration. Assigns the same symbolic revision number to a given revision in all RCS-files. This command is useful for accurately recording a configuration.

Table 6-2: (continued)

Command	Option	Description
`rcsmerge`		Merges two revisions, *rev1* and *rev2*, with respect to a common ancestor. A three-way file comparison determines the parts of lines that are the same in all three revisions, the same in two revisions, or different in all three. Overlapping changes are flagged and reported to the user.
	`-p`	Prints the result of the merged files to the standard output; otherwise, the resulting merged file overwrites the working file.
`rcsstat`		Prints RCS-file status. Prints information about RCS-files, for example, the current version of the file selected with the `-r` option.
`rcstime`		Prints checkin time. Prints the time a particular revision of a given RCS-file was checked in to the system. The revision is selected by number and name, checkin date and time, author, or state.
`rlog`		Reads log messages. Prints the log messages and other information in an RCS-file, for example: RCS-file name, working file name, head (the number of the latest revision on the trunk), default branch, access list, locks, symbolic names, number of revisions and descriptive text.
	`-h`	Prints only the RCS-file name, working file name, head, default branch, access list, locks, symbolic names, and suffix.
	`-t`	Prints the same information as does `-h`, plus the descriptive text.

6.4.1 Placing New Files in an RCS Library

You can use the `ci` command to place new files in a library. The following example assumes that you are in the library's parent directory and want to add the `attr` file to the library:

```
% ci attr
RCS/attr,v  <---- attr
initial revision: 1.1
enter description, terminate with ^D or '.':
>> Orpheus Authoring Tools attr command
>>^D
done
```

The `ci` command creates the RCS-file `attr,v` and stores `attr` in it as revision 1.1. The command prompts you for a description, which should be a synopsis of the contents of the file. All later checkin commands will prompt you for a log entry, which should summarize the changes you made.

You can enter a series of files in a single operation. For example:

```
% ci attr docbld dcb.ch-intro
```

6.4.2 Recording File-Indentification Information with RCS

The RCS system provides a syntax for including **keywords** or **ID markers** in source files to provide file-identification information. An ID marker consists of a keyword enclosed within dollar signs ($). When you retrieve a file from the RCS library, RCS expands the keyword by replacing it with the appropriate information, such as the file name or revision number.

RCS allows you to use keyword markers anywhere in your file as literal strings or comments to identify a revision. For example, if you place the marker `$Header$` into your text file, RCS (with the `co` command) will replace this keyword with the following information:

```
$Header: filename revision_number date time author state$
```

Table 6-3 lists the RCS keywords and their corresponding values.

Table 6-3: RCS ID Keywords

Keyword	Description
`$Author$`	The login name of the user who checked in the revision.
`$Date$`	The date and time the revision was checked in.
`$Header$`	A standard header containing the full pathname of the RCS-file, the revision number, the date, the author, the state, and the locker (if locked).
`$Id$`	Same as `$Header$` except that the RCS-file name is without a path.
`$Locker$`	The login name of the user who locked the revision (empty if unlocked).
`$Log$`	The log message supplied during checkin, preceded by a header containing the RCS-file name, the revision number, the author, and the date. Existing log messages are not replaced; instead, the new log message is inserted after `$Log:...$`.
`$RCSfile$`	The name of the RCS-file without path.
`$Revision$`	The revision number assigned to the revision.

Table 6-3: (continued)

Keyword	Description
$Source$	The full pathname of the RCS-file.
$State$	The state assigned to the revision with the -s option of rcs or ci.

The ident command finds and extracts keywords from any file, even object files and dumps. It searches the files you specify for all occurrences of the pattern $keyword:...$. For example, suppose the C program myfile.c contains the following information:

```
char resid [] = "$Header: Header information$"
```

The command ident will print the following:

```
myfile.c : $Header: Header information$
```

For more detailed information on using keywords in RCS, refer to the co(1) reference page.

6.4.3 Getting Files from an RCS Library

To retrieve a file revision from an RCS-file, check it out of the RCS library by using the co command. The co command retrieves a revision from the RCS-file and stores it in a corresponding working file.

Revisions of an RCS-file can be checked out locked or unlocked. Locking a revision prevents overlapping updates. When you check out a file for reasons other than editing (reading or processing, for example), the revision need not be locked. (A revision checked out for editing and later checkin must normally be locked.) For example:

```
% cd /usr/projects/dcb_tools
% co -u attr
RCS/attr,v ----> attr
revision 1.6 (unlocked)
done
```

This command creates a copy of the most recent version of the RCS-file (with keyword information included) and places it in your current directory (/usr/projects/dcb_tools in this example). The –u option prevents RCS from locking the file. To get a copy of any earlier version, use the –r option. For example, to retrieve version 1.5 of a file that is now at version 1.8, you would use a command like the following:

```
% cd /usr/projects/dcb_tools
% co -r1.5 attr
RCS/attr,v ----> attr
revision 1.5
done
```

You can also retrieve a series of files with a single `co` command. For example:

```
% co attr unstamp
RCS/attr,v ----> attr
revision 1.5
done

RCS/unstamp,v ----> unstamp
revision 1.2
done
```

6.4.4 Checking Edited Files Back into an RCS Library

To replace one or more edited files, use the `ci` command. This command places the contents of each working file in the corresponding RCS-file. Normally, RCS checks whether the revision to be deposited in the library is different from the preceding one, and alerts the user.

Also, because the `ci` command deletes your working files during checkin, you may want to use either the `–l` option or the `–u` option to preserve your working files by performing an implicit checkout operation. This is desirable if you want to save the current revisions and continue editing.

6.4.5 Working with Multiple Versions of Files

Section 6.2 provides an overview of branching concepts in a version control system. The following discussion provides specific examples that illustrate how RCS handles branching of multiple files.

RCS arranges file revisions in a tree of deltas. Each file in a revision tree contains the following kinds of information: a revision number, a checkin time and date, the author's identification, a log entry, a state, and the actual text. All of these file attributes are determined at the time the revision is checked in to the library. The "state" attribute indicates the status of the revision, which is set to "experimental" during checkin, but which can be later changed to "stable" or "released."

The `ci` command creates a revision tree with a root revision that is normally numbered 1.1. Unless you specify a revision number explicitly, `ci` assigns new revision numbers by incrementing the level number of the previous

revision (1.2, 1.3, 1.4, and so on). To begin a new release, use the following command:

```
% ci -r2.1 unstamp
```

or

```
% ci -r2 unstamp
```

This action assigns the number 2.1 to the new revision. Checking in the file to the library without the `-r` option automatically assigns the numbers 2.2, 2.3, and so on, to the later revisions of the file.

Suppose two development teams begin development on separate releases of the `unstamp` command, beginning from revision number 1.2. At this point, both teams can check out the latest revision by using the `co` command with the `-l` option as follows:

```
% co -l unstamp
```

After editing the file, the first team can check in the file by using the `ci` command, and will be alerted by RCS that the new revision number is 1.3. For example:

```
% ci unstamp
RCS/unstamp,v  <---- unstamp
new revision 1.3; previous revision 1.2
enter log message:
(terminate with a ^D or single '.')
>> Changed defaults check.
>>^D
done
```

However, if the second team tries to check in the file with the same action, RCS will issue the following message:

```
RCS/unstamp,v <----- unstamp
ci error: no lock set by user-name
```

At this point, the second team can create a branch by using the `ci` command as follows:

```
% ci -r1.3.1 unstamp
```

This action will result in a branch with revision number 1.3.1.1. To continue development along this branch, the second team should use the current branch revision number on all subsequent checkouts of the file. For example:

```
% co -r1.3.1.1 unstamp
```

Creating new branches in RCS is accomplished through the use of the `ci` command; to continue development along a particular branch, use the `-r` option with the `co` command.

The preceding discussion describes how RCS handles revisions of individual files; the system also allows you to work with groups (or sets) of files that you specify. See Section 6.4.8 for more information on working with file configurations in RCS.

6.4.6 Displaying Differences in RCS Files

You can examine an RCS-file for differences between versions with the `rcsdiff` command.

The `rcsdiff` command runs `diff`(1) to compare two revisions of each RCS-file given. For example, to find the differences between the latest version of the `attr` file (1.8, being edited to become 1.9) and the immediately preceding version (1.7), you would use the following command:

```
% rcsdiff -r1.7 attr
===================================
RCS file: RCS/attr,v
retrieving revision 1.7
diff -r1.7 attr
31d30
< # and is version linked to the docbld command
```

To check the differences between versions 1.3 and 1.4 of the `attr` file, you would use the following command:

```
% rcsdiff -r1.3 -r1.4 attr
===================================
RCS file: RCS/attr,v
retrieving revision 1.3
retrieving revision 1.4
diff -r1.3 -r1.4
5a6
< uts=-04
---
> uts=-05
```

6.4.7 Reporting Revision Histories of RCS Files

Use the `rlog` command to examine the revision history of a file. For example, the `rlog` command provides you with the following detailed information:

```
% rlog unstamp
RCS file:        RCS/unstamp,v;  Working file:     unstamp
head:            1.2
branch:
locks:           ;  strict
access list:
symbolic names:
comment leader:  "#  "
total revisions: 2;     selected revisions: 2
description:
unstamp source file

----------------------------
revision 1.2
date: 92/06/09 15:51:16;  author:gunther;  state:Exp;  lines
added/del:
Fixed copyright notice

----------------------------
revision 1.1
date: 92/06/09 15:49:16;  author:gunther;  state:Exp;
Initial revision
```

Note the type and amount of information that is available to you using the `rlog` command. RCS prints the following information for each RCS-file:

- RCS-file name
- Working file name
- Head (the number of the latest revision on the trunk)
- Default branch
- Access list
- Locks on the file
- Symbolic names (if any)
- Suffix
- Total number of revisions
- Number of revisions selected for printing
- Descriptive text

This information is followed by entries for the selected revisions in reverse chronological order for each branch. If entered without specifying options, `rlog` prints complete information for the file you select. See the `rlog`(1) reference page for more information on using options to restrict the output of the `rlog` command.

6.4.8 Configuration Control Concepts

A **configuration** in RCS refers to a group or set of file revisions, in which each revision comes from a different file revision group. File revisions can be selected (checked out) according to certain criteria. You can check out sets of files from an RCS library based on the following selection criteria:

- Default selection

  RCS lets you choose the latest revision of all files by default. For example, the following command retrieves the latest revision on the default branch of each RCS-file in the library:

  ```
  % co *,v
  ```

- Release-based selection

  You can also specify a release or branch number to select the latest revision in that release or branch. For example, the following command retrieves the latest revision with release number 2 from each RCS-file:

  ```
  % co -r2 *,v
  ```

- State and author-based selection

  RCS allows you to select files according to state attributes. For example, suppose you want to retrieve the latest revision with release number 2 whose state attribute is ''Released.'' This can be accomplished by issuing the following command:

  ```
  % co -r2 -sReleased *,v
  ```

  You can also select a revision by author, by using the `-w` option.

- Date-based selection

  Revisions can also be selected by date. Suppose a release of an entire system was completed and current as of June 15, at 2:00 p.m. The following command checks out all files of that release, with the `-d` option specifying the cutoff date as June 15:

  ```
  % co -d "June 15,2:00 pm" *,v
  ```

- Name-based selection (using symbolic names)

  RCS allows you to assign **symbolic names** to revisions and branches. In large systems and development efforts, a single release number or date may not be sufficient to collect all the appropriate revisions from all groups.

  For example, suppose you need to combine release 2 of one subsystem with release 10 of another. Most likely, the creation dates of these revisions will be different, so passing a single revision number or date to the `co` command will not be appropriate in this case. Using symbolic

revision names can help solve this problem; each RCS-file can contain a set of symbolic names that are mapped to the numeric revision numbers.

For instance, suppose you set the symbolic name IFT2 to release number 2 in the file `attr,v` and to revision number 10.2 in `unstamp,v`. In this case, a single `co` command retrieves the latest revision of release 2 from `attr,v` and revision 10.2 from `unstamp,v` as follows:

```
% co -rIFT2 attr,v unstamp,v
```

The `rcsfreeze` command can be used to assign a symbolic revision name to a set of RCS-files that form a configuration. To assign a unique symbolic revision name to the most recent revision of each RCS-file of the main trunk, use the `rcsfreeze` command each time a new version is checked in. For more information on assigning symbolic names to RCS-files, refer to the `rcsfreeze`(1) reference page.

For large software development efforts, the ability to retrieve all revisions with one command makes configuration management an organized and efficient task.

6.5 Using SCCS

The SCCS system is composed of several independent commands, each of which can be used independently. The `sccs` command is a unified interface that simplifies the usage of the most common SCCS commands and provides several additional functions by combining the operations of multiple commands. It does not support all of the functions of the individual commands.

Each form of the `sccs` command includes the keyword `sccs` and the name of one function, such as `edit`, followed by options and the names of the file or files to be manipulated. Table 6-4 lists the `sccs` commands. The sections following the table provide more information on some of these commands. In these discussions, command options are omitted except where required. Commands that are also individual (''low level'') commands are indicated in the table. The complete list of individual commands is summarized in Table 6-9; for detailed information on their use, along with descriptions of their options, refer to their individual reference pages.

Table 6-4: Summary of sccs Command Functions

Command Name	Low Level	Description
`admin`	Yes	Creates an s-file or changes some characteristic of an existing s-file.

Table 6-4: (continued)

Command Name	Low Level	Description
`check`	No	Reports on files being edited and the names of the users editing them. Differs from `info` in that `check` returns a meaningful exit status and displays no report if no files are being edited.
`clean`	No	Removes from your directory all files that can be regenerated from the named s-file.
`create`	No	Creates an s-file without removing the g-file.
`deledit`	No	Performs a `delta` operation followed by an `edit` operation on the same file.
`delget`	No	Performs a `delta` operation followed by a `get` operation on the same file.
`delta`	Yes	Checks an edited g-file back into the library, recording the changes made and their history. Removes the p-file.
`diffs`	No	Compares a g-file that is checked out for editing with an earlier version reconstructed from the s-file.
`edit`	No	Checks out an s-file for editing; regenerates the g-file and places it in your directory. Creates a p-file.
`fix`	No	Removes the most recent delta and presents the g-file for reediting. Same as entering `rmdel` followed by `edit`.
`get`	Yes	Regenerates a g-file, usually but not always for a purpose other than editing. (The `sccs edit` command, which duplicates the function of `sccs get -e`, is the usual way to regenerate a g-file for editing.)
`help`	Yes	Given a command name or an SCCS message number, displays information about that item. (The individual command's form is `sccshelp`.) Each SCCS message has an identification code; for example, the "no ID keywords" message's code is `cm7`. The `sccs help cm7` command displays a description of this error. The `sccshelp delta` command returns a syntax diagram for the `delta` command.
`info`	No	Reports on files being edited and the names of the users editing them.
`print`	No	Displays the revision histories of the named file or files, then displays the SCCS file, with ID information added to the beginning of each line.
`prs`	Yes	Displays the revision histories of the named file or files.
`prt`	No	Same as `prs`.

Table 6-4: (continued)

Command Name	Low Level	Description
`rmdel`	Yes	Removes the most recent delta from the specified branch of a named s-file.
`sccsdiff`	Yes	Compares two versions of the s-file. Requires explicit specification of the s-file name.
`tell`	No	Reports on files being edited. Differs from `info` in that only file names are reported.
`unedit`	No	Aborts editing of a g-file. Deletes the p-file, releasing the s-file so that other users can check it out. If the g-file is present in your working directory, `sccs unedit` removes it and performs a `get` command on the s-file; if no g-file is present, no `get` command is executed. (Equivalent to the `unget` command.)
`what`	Yes	Searches a file for an SCCS ID pattern and displays the text that follows it.

6.5.1 Placing New Files in an SCCS Library

You can use the `sccs create` command to place new files in a library. The following example assumes that you are in the library's parent directory and want to add the `attr` file to the library:

```
% sccs create attr
attr:
1.1
141 lines
```

Do not include the prefix `s.` in the file name you specify; SCCS applies it automatically.

You can enter a series of files in a single operation. For example:

```
% sccs create attr docbld dcb.ch-intro
```

After creating the s-file in the library, the `sccs create` command adds a comma as a prefix to the name of the original file; for example, `attr` becomes `,attr`. This action preserves the original g-file with its keywords (if any) unexpanded. Then SCCS fetches a copy of the file by using a `get` command; this fetched version is ready for distribution.

You can also insert files in the library with the `sccs admin -i` command, using the following syntax:

sccs admin –i[*path/*] *input-file* [*path/*] *s-filename*

For example:

```
% sccs admin -iunstamp unstamp
```

The name *path/input-file* specifies the input file. Regardless of the name of this file, the s-file will be named `s.`*s-filename*. Do not include any white space between the `-i` option and *path/input-file*. Do not include the prefix `s.` in *s-filename*; SCCS applies it automatically. Using the `admin -i` command avoids the renaming of the original g-file and the fetching of a version with expanded keywords. See Section 6.5.8 for more information on using the `admin` command.

You can use the `admin -i` option to enter several files with a short shell script; the following command-line example is implemented in `csh`:

```
% foreach x (attr docbld dcb.ch-intro)
? sccs admin -i$x $x
? end
```

6.5.2 Recording File-Identification Information with SCCS

The SCCS system provides a syntax for including ID keywords in source files to provide file-identification information. An ID keyword consists of a single letter enclosed within percent signs (`%`). When you retrieve a file for any purpose other than editing, SCCS expands each ID keyword by replacing it with the appropriate information, such as the SID or the file name. Table 6-5 lists the SCCS ID keywords.

Table 6-5: SCCS ID Keywords

Keyword	Description
`%B%`	The branch number of a retrieved g-file
`%C%`	The current line number in the file, intended to identify messages output by a program
`%D%`	The retrieval date of a g-file retrieved by a `get` command, in the form *yy/mm/dd*
`%E%`	The creation date of a delta, in the form *yy/mm/dd*
`%F%`	The file name of the s-file
`%G%`	The creation date of a delta, in the form *mm/dd/yy*
`%H%`	The retrieval date of a g-file retrieved by a `get` command, in the form *mm/dd/yy*
`%I%`	The highest SID applied to the file retrieved
`%L%`	The level number of a retrieved g-file
`%M%`	The current module (file) name; for example, `prog.c`

Table 6-5: (continued)

Keyword	Description
%P%	The full pathname of the s-file
%Q%	The value of the q flag in the s-file
%R%	The release number of a retrieved g-file
%S%	The sequence number of a retrieved g-file
%T%	The retrieval time of a g-file retrieved by a get command, in the form *hh:mm:ss*
%U%	The creation time of a delta, in the form *hh:mm:ss*
%W%	A shorthand for %Z%%M% Tab %I%
%Y%	A placeholder for the value of the t flag (set by the admin command); not meaningful to SCCS itself
%Z%	A placeholder that expands to the string @(#) for the what command to find

SCCS handles ID keywords anywhere in a file. The purpose of the SCCS what command is to find and display expanded ID keywords in a file. The what command searches for lines containing the string @(#), which is generated by the %Z% keyword or the %W% shorthand keyword, and displays those lines. For example:

```
% what /usr/bin/attr
/usr/bin/attr:
        attr 1.8 of 4/15/92
```

The line displayed in this example is part of a shell script and was coded as follows:

```
# SCCSID: %Z%%M% %I% of %G%
```

If your file does not contain ID keywords, SCCS reports that fact when you put the file in the library and when you retrieve it. You can set the file's i flag to specify that this condition will be a fatal error. (See Section 6.5.8 for a description of file flags.) The purpose of the i flag is to prevent a delta command from merging a g-file with expanded keywords (or with no keywords) with the s-file.

6.5.3 Getting Files from an SCCS Library

There are two reasons to get files from an SCCS library: for any use except editing, such as distribution, or for editing.

You can edit a file as part of the straight-line progress of its version history, or you can create a branching tree. Section 6.5.5 describes how to create a tree wherein multiple parallel versions are stored together in the same s-file.

6.5.3.1 Getting Files for Purposes Other Than Editing

For any use except editing, you get SCCS files with the `sccs get` command. For example:

```
% cd /usr/projects/dcb_tools
% sccs get attr
1.8
126 lines
```

This command creates a copy of the most recent version of the s-file with SCCS keywords expanded (see Table 6-5) and places it in your current directory (`/usr/projects/dcb_tools` in this example). To get a copy of any earlier version, use the *–rSID* option. For example, to retrieve version 1.5 of a file that is now at version 1.8, you would use a command like the following:

```
% cd /usr/projects/dcb_tools
% sccs get -r1.5 attr
1.5
128 lines
```

See Section 6.5.5 for information on managing more complex trees of SCCS files.

You can use the –p option to retrieve a file and write it to standard output instead of implicitly creating a g-file with the same name as the s-file. See the `get`(1) reference page for more information.

6.5.3.2 Getting Files for Editing

To edit a file, check it out of the library with the `sccs edit` command. For example:

```
% sccs edit attr
1.8
new delta 1.9
126 lines
```

This command creates a copy of the most recent version of the s-file with SCCS keywords unexpanded (see Table 6-5) and places it in your directory for editing. The command also creates a p-file identifying the person who checked out the file for editing.

You can check on the status of files with the `sccs info` command. For example:

```
% sccs info
unstamp: is being edited: 1.4 1.5 gunther 92/09/07 10:42:19
```

You can also use the `get -e` command to retrieve a file for editing.

6.5.3.3 Managing Multiple Files and New Releases

You can retrieve a series of files with a single `get` or `edit` command. For example:

```
% sccs get attr unstamp
SCCS/s.attr:
1.8
126 lines

SCCS/s.unstamp:
1.2
55 lines
```

If you specify the name `SCCS` instead of one or more file names, SCCS retrieves every s-file in the library.

To create a new release of a file, you fetch it using the `-r` option to specify the new release number in the `sccs edit` command. For example, the following command initiates Release 2 of the `docbld` file:

```
% sccs edit -r2 SCCS
SCCS/s.docbld:
1.50
new delta 2.1
1042 lines

SCCS/s.dcb_defaults:
1.50
new delta 2.1
63 lines

SCCS/s.dcb_diag.sed:
1.50
new delta 2.1
188 lines
```

6.5.4 Checking Edited Files Back into an SCCS Library

To replace in the library a file you have edited, use the `sccs delta` command. SCCS will prompt you for a comment. For example:

```
% sccs delta attr
Comments? (^D to end)
Changed defaults check.  Now looks only for "flc="
Ctrl/D
1.9
4 inserted
4 deleted
124 unchanged
```

If you specify the name SCCS instead of one or more file names, SCCS performs a `delta` on every s-file in the library. Coupled with a similar `edit` command, this function is useful for sets of files that must be kept in version synchronization even when not all of them are edited. SCCS asks for comments only once and applies the same comment to each file.

The `sccs delget` and `sccs deledit` commands perform a `delta` followed by a `get` or an `edit` operation respectively.

6.5.5 Working with Multiple Versions of Files

Section 6.2 provides an overview of branching concepts in a version control system. The following section provides specific examples that illustrate how SCCS handles branching of multiple versions of files.

Suppose two development teams begin development on separate versions of the `unstamp` file, beginning from SID 1.2. To enable branching, run the `sccs admin -fb` command as follows:

```
% sccs admin -fb unstamp
```

The first team uses an `edit` command to create version 1.3 as follows:

```
% sccs edit unstamp
1.2
new delta 1.3
55 lines
```

The second team uses an `edit -b` command to create a branch as follows:

```
% sccs edit -b unstamp
1.2
new delta 1.2.1.1
55 lines
```

Consider now a tree for the `unstamp` file with a main trunk and branches numbered 1.2.1, 1.2.2, and 1.3.1. To get the latest version from branch 1.2.2 for distribution, you would use the following command:

```
% sccs get -r1.2.2 unstamp
1.2.2.1
55 lines
```

As an SCCS tree becomes more complex, ensuring that you have the latest delta for editing can become cumbersome because you must know the delta you want to retrieve. You can use the `–t` option to the `sccs get` and `sccs edit` commands to specify the absolute latest delta regardless of its SID.

You can merge a branched SCCS file back into the main trunk by using the `sccs edit -i` command and by specifying the version or versions you want to merge. For example, the following command creates version 1.5 of the `unstamp` command, including all the deltas in the range from 1.2.1.1 to 1.2.1.3. The deltas are correlated so that the result is the accumulation of all specified changes.

```
% sccs edit -i1.2.1.1-1.2.1.3 unstamp
Included:
1.2.1.1
1.2.1.2
1.2.1.3
1.4
new delta 1.5
55 lines
```

6.5.6 Displaying Differences in SCCS Files

You can examine an SCCS file for differences between versions with either the `sccs diffs` command or the `sccsdiff` command, depending on what forms of the file are available.

The `sccs diffs` command compares the g-file with the specified version of the s-file. For example, to find the differences between the latest version of the `attr` file (1.8, being edited to become 1.9) and the immediately preceding version (1.7) you would use the following command:

```
% sccs diffs -r1.7 attr

------- attr -------
31d30
< #     and is version-linked to the docbld command
```

To check the differences between versions 1.3 and 1.4 of the `attr` file, you would use the following command:

```
% sccs sccsdiff -r1.3 -r1.4 SCCS/s.attr
< uts=-04
---
> uts=-05
```

As this example shows, you can enter a pathname for the s-file itself. Because of this design, you can use this command from any directory instead of having to change to the directory containing the SCCS library.

6.5.7 Reporting Revision Histories of SCCS Files

Use the `sccs prs` command to examine the revision history of a file. For example:

```
% sccs prs unstamp
SCCS/s.unstamp:

D 1.2 92/09/20 11:23:36 gunther 2 1      00000/00006/00055  1
MRs:                                  2
COMMENTS:                                3
Fixed copyright notice


D 1.1 92/09/19 09:39:11 gunther 1 0      00061/00000/00000
MRs:
COMMENTS:
date and time created 92/09/19 09:39:11 by gunther
```

The `D`, `MRs`, and `COMMENTS` keywords indicated by callouts in this display are part of the complete set of SCCS keywords. Use the `sccs help` command to display a list of the keywords and their meanings. The `D` keyword (callout 1) marks delta information. The two numbers after `gunther` (the programmer's user name) indicate the new and old revision levels. The slash-separated numbers indicate the numbers of lines added, deleted, and left unchanged. The keyword `MRs` (callout 2) lists major revisions; the major revision is the first element of a file's SID.

Use the `sccs get -m` command to retrieve a copy of the file with SID numbers added as a prefix to each line. A file retrieved in this way shows you what delta produced every line in the retrieved version. Keep in mind that a given delta can be overlaid by later deltas; you might need to use the `-r` option to find particular changes.

6.5.8 Performing Administrative Functions

The `sccs admin` command performs several administrative functions. Each function is specified by an option to the `admin` command, as described in Table 6-6.

Note

Your system administrator can set permissions so that only the administrator can use the `admin` command.

Table 6-6: SCCS admin Command Options

Option	Description
`–auser s-file`	Adds the specified user to the list of users allowed to make changes to the named s-file. The user name can be a group ID; all users in that group are added.
`–dflag s-file`	Turns off (deletes) the named flag in the s-file.
`–euser s-file`	Removes the specified user from the list of users allowed to make changes to the named s-file. The user name can be a group ID; all users in that group are removed.
`–fflag s-file`	Turns on the named flag in the s-file.
–h *s-file*	Checks the structure of the named s-file and compares a newly computed checksum with the checksum that is stored in the s-file. This option helps you detect both accidental damage and damage caused by modifying SCCS files directly with non-SCCS commands.
`–iinput-file s-file`	Creates `SCCS/s.s-file`, using *input-file* as the initial contents. Differs from `sccs create` in that `admin –i` does not rename the g-file or fetch a copy of the s-file; the g-file is left untouched in your directory.
`–mMR-list s-file`	Specifies a list of Modification Request (MR) numbers to be inserted into the SCCS file as the reason for creating the initial delta.
`–ns-file`	Creates an empty s-file.
`–rSID s-file`	Specifies the initial SID when creating an s-file.
`–tfile s-file`	Adds the contents of *file* to the s-file, flagging it as added text. If *file* is omitted, any such added text is deleted. Useful for including documentation to ensure its distribution with the s-file.

Table 6-6: (continued)

Option	Description
–y"*comment*" *s-file*	Inserts the comment text in the initial delta in a manner identical to the workings of the `delta` command. The default comment, if the –y option is not used, is a line giving the date and time of the file's creation and the name of the user who created it.
–z *s-file*	Recomputes the s-file's checksum in case the file has been corrupted.

Caution

Using the `val` and `admin -z` commands to repair damaged s-files is risky and should be left to your system administrator or to a designated SCCS librarian.

The flags for the `admin –f` and `admin –d` options are described in Table 6-7.

Table 6-7: Flags for the admin Command

Flag	Description
b	Allows branches to be made using the –b flag to the `edit` command.
c*SID*	Specifies *SID* as the highest delta that a `get –e` command can use.
d*SID*	Specifies the default SID to be used on a `get` or `edit` command.
f*SID*	Specifies *SID* as the lowest delta that a `get –e` command can use.
i	Causes the "no Id keywords" error message to be a fatal error rather than a warning.
j	Permits editing of the s-file by more than one person concurrently.
l*SID*[,*SID..*]	Locks the specified *SID*s from being retrieved for editing. You can lock all deltas with the –fla flag, and you can unlock specific deltas with the –d flag.

Table 6-7: (continued)

Flag	Description
m*name*	Substitutes *name* for all occurrences of the %M% keyword when keywords are expanded by a `get` command. The default name is the s-file's name without the `s` prefix.
n	Causes the `delta` command to create a null delta in any releases that are skipped when a delta is made in a new release. For example, if you make delta 5.1 after delta 2.7, releases 3 and 4 will be null. The resulting null deltas can serve as points from which to build branch deltas. Without this flag, skipped releases do not appear in the s-file.
q"*text*"	Substitutes *text* for all occurrences of the %Q% keyword when keywords are expanded by a `get` command.
t*type*	Substitutes *type* for all occurrences of the %Y% keyword when keywords are expanded by a `get` command.
v[*program*]	Makes delta prompt for Modification Request (MR) numbers as the reason for creating a delta. The name *program* specifies the name of an MR number validity-checking program. (See the `delta`(1) reference page.)

For example, the following command uses the contents of an existing text file to create an s-file beginning at SID 2.1 and identified with a comment. The s-file's `i` flag is set. The command places the resulting s-file in the SCCS library under the user's working directory.

```
% sccs admin -iunstamp -fi -r2 -y"Initial release" unstamp
```

This example does not destroy the original file.

6.5.9 Using SCCS Options

The `sccs` command supports the options listed in Table 6-8. Note that these options must include the SCCS function command keyword as shown in the examples in the table. Do not include any space between the options and their arguments.

Table 6-8: SCCS Command Options

Option	Description

–d*dirname*	Specifies a directory to use as the SCCS library's parent. Allows access to SCCS libraries without requiring that your working directory be the parent. For example:
	`% pwd`
	`/usr/users/gunther`
	`% sccs -d/usr/src/dcb_tools get attr`
	`1.8`
	`126 lines`

–p*path*	Adds `path` to the final element of the pathname for the file you specify. By default, SCCS adds `SCCS` so that the path specified in the –d example resolves to `/usr/src/dcb_tools/SCCS/s.attr`. If your SCCS library is not named `SCCS`, use the –d option to modify this component of the path.
–r	Runs with the real user's UID instead of changing to the `sccs` UID. For security purposes, SCCS normally sets the ownership of files in an SCCS library so that they belong to the `sccs` UID. This option is useful if you are using SCCS to manage a library for yourself only; you can create the `SCCS` directory with normal permissions, and the –r option will cause SCCS to manipulate files therein using your own UID.

6.5.10 Summary of Individual SCCS Commands

Table 6-9 provides a brief description of the individual SCCS commands. Note that some of these commands are not supported by the `sccs` command. Refer to the appropriate command's reference page for more detailed information.

Table 6-9: Individual SCCS Commands

Command Name	Supported by sccs Command	Description
`admin`	Yes	Creates an s-file or changes some characteristic of an existing s-file.
`cdc`	No	Changes the comments associated with a delta.

Table 6-9: (continued)

Command Name	Supported by sccs Command	Description
`comb`	No	Combines two or more consecutive deltas of an s-file into a single delta. Combining deltas can reduce storage requirements.
`delta`	Yes	Checks an edited g-file back into the library, recording the changes made and their history. Removes the p-file.
`get`	Yes	Gets a specified version of an s-file. Use this command to get a copy of a file to edit or compile. For editing, use the `get -e` command, which checks out an s-file for editing, regenerates the g-file and places it in your directory, and creates a p-file.
`prs`	Yes	Displays the revision histories of the named s-file or s-files.
`rmdel`	Yes	Removes the most recent delta from the specified branch of a named s-file.
`sccsdiff`	Yes	Compares two versions of the s-file. Requires explicit specification of the s-file name.
`sccshelp`	No	Provides an explanation of a diagnostic message or of an SCCS command name.
`unget`	No	Removes the effect of a previous use of the `get -e` command by deleting the p-file and replacing the g-file with a copy having its ID keywords expanded. (Equivalent to the `sccs unedit` command.)
`val`	No	Computes a checksum on an s-file to see if the result matches the checksum stored in the file. Use this command with the `sccs admin -z` command to detect and repair corrupted files.
`what`	Yes	Searches a file for an SCCS ID pattern and displays the text that follows it. Use this command to find identifying information describing the source versions (kept under SCCS control) used to construct a program.

Caution

Using the `val` and `admin -z` commands to repair damaged s-files is risky and should be left to your system administrator or to a designated SCCS librarian.

6.6 Functional Comparison of RCS and SCCS Commands

Table 6-10 provides a brief comparison of the operations of RCS and SCCS and the commands that are used to achieve similar functions. Refer to the reference pages for detailed information on using the individual commands.

Table 6-10: Functional Comparison: RCS and SCCS Commands

Tasks	RCS Command	SCCS Commands
Create a new file from your original.	`ci` *file*	`sccs create` *file* `sccs admin -i`*sfile gfile* `admin -i`*path/sfile gfile*
Get a copy of a file with expanded keywords.	`co -u` *file*	`sccs get` *file* `get` *file*
Get a copy of a file with unexpanded keywords.		`sccs get -k` *file* `get -k` *file*
Check out a file.	`co -l` *file*	`sccs edit` *file* `get -e` *file*
Check in an edited file.	`ci` *file*	`sccs delta` *file* `delta` *file*
Show revision histories of a file.	`rlog` *file*	`sccs prs` *file* `prs` *file*
Examine differences between file revisions.	`rcsdiff -r`*rev file*	`sccs diffs -r`*rev file* `sccsdiff -r`*rev -r rev file*
Merge file revisions.	`rcsmerge -r`*revs file*	`sccs edit -i`*revs file*
Find identifying information.	`ident`	`sccs what` `what`

Table 6-10: (continued)

Tasks	RCS Command	SCCS Commands
Perform administrative tasks.	`rcs`	`admin`
Clean up your directory. (Remove unchanged files.)	`rcsclean`	`sccs clean`

Building Programs with the make Utility 7

The make utility builds up-to-date versions of programs. It is most useful for large programming projects in which multiple source files are combined to form a single program or for building a set of programs that are parts of a single product or application.

The make utility works by comparing the creation date of a program to be built, called the **target** or **target file**, with the dates of the files that make it up, called **dependency files** or **dependents**. If any of a given target's dependents are newer than the target, make considers that the target is out of date. In this case, make rebuilds the target by performing the necessary compiling, linking, or other steps. Each dependent can also be a target; for example, an executable program is made from object modules, which are in turn made from source files. Dependents that are newer than the target are called **younger** files.

The make command accepts options to control or modify how the building process is performed. The make utility does not address the problem of maintaining more than one version of the same source file.

Using the make utility to maintain programs, you can do the following:

- Combine the instructions for creating a large program in a single file
- Define macros to use within the make description file
- Use shell commands
- Create or update libraries
- Include files from other programs

The DEC OSF/1 system provides several versions of the make command, with differences in features among versions:

- make(1)

  The default version of make.

- make(1u)

  Offers enhanced functionality over make(1).

- make(1p)

  Provided for POSIX compliance.

The make(1) and make(1u) versions are included in the base operating system subsets. The make(1p) version is included in the "Software Development Environment (Software Development)" subset.

Refer to the respective reference pages for further information. This chapter describes only the features of the make(1) version.

7.1 Operation of the make Utility

The make utility uses the following sources of information:

- A description file that you create
- File names
- Time stamps of the files from the file system
- A set of rules that tell make how to build files

The make utility depends on files' time stamps. For make to work properly on a distributed system, the date and time on all systems in the network must be synchronized.

The make utility creates a target file using the following step-by-step procedure:

1. Finds the name of the target file in the description file
2. Finds a line that describes the dependents of the target, called a **dependency line**
3. Ensures that all the target's dependency files exist and are up to date
4. Determines if the target is current with respect to its dependents
5. If the target or one of the dependents is out of date, creates the target by one of the following methods:
   - Executes commands from the description file
   - Uses internal rules to create the file (if they apply)
   - Uses default rules from the description file

If all files described on the dependency line are up to date when make is run, make indicates that the target is up to date and then stops. If any dependents are newer than their targets, make re-creates only those targets that are out of date. Any missing files are deemed to be out of date. If a given target has no dependents, it is always out of date, and make rebuilds it every time you run make. The make process works from the top down in determining what targets need to be rebuilt and from the bottom up in the actual rebuilding stage.

When the make utility runs commands to create a target, it replaces macros with their values, echoes each command line to the standard output, and then

runs the command. (See Section 7.2.9 for information about macros.) The make utility runs commands that it can execute directly, such as rm or cc, without invoking a new shell. The utility invokes each command line that includes shell functions, such as pipes or redirection, in a new shell.

You start the make utility in the directory that contains the description file. The syntax of the make command is as follows:

make [**–f** *makefile*] [*options*] [*targets*] [*macro definitions*]

The make utility examines the command-line entries to determine what to do. First, it assigns values for the macro definitions on the command line (entries containing equal signs), if there are any, and for the macro definitions in the description file. If there is a definition on the command line for a macro name that is also defined in the description file, make uses the command-line definition and ignores the definition in the description file.

Next, make looks at the options. Refer to the make(1) reference page for a complete list of the options that make supports.

The make utility interprets the remaining command line entries as the names of targets. It processes the targets in left-to-right order. If there are no targets on the command line, make processes the first target named in the description file and then stops.

7.2 Description Files

The description file tells make how to build the target by defining what dependencies are involved and what their relationships are to the other files in the procedure. The description file contains the following information:

- Definitions of macros in the description file
- One or more target names
- Dependency file names that make up the target files
- Commands that create the target files from the dependents
- Any of the pseudotargets .DEFAULT, .IGNORE, .PRECIOUS, .SILENT, or .SUFFIXES

  These identifiers are called pseudotargets because they are not real targets. They are built-in names that make interprets in special ways. For example, the .SILENT pseudotarget instructs make not to echo command lines as it runs them. Do not use any of these names for a real target.

The make utility determines what files to create to get an up-to-date copy of the target by checking the dates of the dependency files. If any dependency file was changed more recently than the target, make creates all the files that

are affected by the change, including the target. In most cases, the description file is easy to write and does not change often.

The `make` utility normally looks for a description file named either `makefile` or `Makefile`. If you name the description file `makefile` or `Makefile` and are working in the directory containing that description file, you enter the `make` command without any options or arguments to bring the first target and its dependency files up to date, regardless of the number of files that were changed since the last time `make` created the target file. You can override the default file name by using the `–f` option to the `make` utility to specify the name of the desired description file, as in the following example:

```
% make -f my_makefile
```

This option allows you to keep several description files in the same directory.

7.2.1 Format of a Description File Entry

The general format of a description file entry is as follows:

target1 [*target2...*] **:**[**:**] [*dependent...*] [**;** *commands*] [**#** *comment...*]

The items inside brackets are optional. Targets and dependents are file names (strings of letters, numbers, periods, and slashes). The `make` command recognizes wildcard characters, such as asterisks (`*`) and question marks (`?`). Each line in the description file that contains a target name is called a dependency line. The dependency line is followed by one or more command lines that specify the process steps to create the target.

Because `make` uses the dollar sign (`$`) to designate a macro, you must not use this character in file names of targets and dependencies. Similarly, do not use the dollar sign in commands in the description file unless you are referring to a defined `make` macro. (Macros are described in Section 7.2.9, Section 7.2.10, and Section 7.2.12.)

To place comments in the description file, use a number sign (`#`) to begin the comment text. The `make` utility ignores the number sign and all characters on the same line after the number sign. The `make` utility also ignores blank lines.

You can enter lines that are longer than the line width of the input device by putting a backslash (`\`) at the end of the line that is to be continued. Do not extend comment lines in this way; begin each new comment line with its own number sign.

7.2.2 Using Commands in a Description File

A command is any string of characters, not including a number sign or a newline character. Commands can appear after a semicolon (;) on a dependency line or on lines immediately following a dependency line. Each command line after the dependency line must begin with a single tab character.

When you define command sequences for the targets in the description file, either specify one command sequence for each target or specify separate command sequences for special sets of dependencies.

To use one command sequence for every use of the target, use a single colon (:) following the target name on the dependency line. For example, the following lines define a target, `test` with a set of dependency files and a set of commands to create the target:

```
test:  dependency list1...
        command list...
   .
   .
   .
test:  dependency list2...
```

As shown here, a target name can appear in several places in the description file with different dependency lists, but there can be only one command list associated with the target name. The `make` utility finds all the dependency lines for a given target and concatenates all their dependency lists into a single list. When any of the dependents has been changed, `make` can run the commands in the one command list to create the target.

To specify more than one set of commands to create a particular target file, enter more than one dependency definition. Each dependency line must have the target name followed by two colons (: :), a dependency list, and a command list that `make` uses if any of the files in the dependency list changes. For example, the following lines define two separate processes to create the target file `test`:

```
test:: dependency list1...
        command list1...
   .
   .
   .
test:: dependency list2...
        command list2...
```

If any of the files in dependency `list1` changes, `make` runs command `list1`; if any of the files in dependency `list2` changes, `make` runs command `list2`. To avoid conflicts, a given dependency file cannot appear in both dependency `list1` and dependency `list2`.

Note

Because `make` runs the commands on each command line independently of preceding or subsequent command lines, be careful when using certain commands (for example, `cd`). In the following example, the `cd` command has no effect on the `cc` command that follows it:

```
test:  test.o
        cd /u/tom/newtest
        cc main.o subs.o -o test
```

To make the `cd` command affect the `cc` command, place both commands on the same line, separated by a semicolon. For example:

```
test:  test.o
        cd /u/tom/newtest; cc main.o subs.o -o test
```

You can simulate a multiline shell script by using backslashes on continued lines:

```
test:  test.o
        cd /u/tom/newtest; \
        cc main.o subs.o -o test
```

This example works exactly the same as the one immediately before it. Note that each line continued with a backslash (the `cd` line in this example) must have a semicolon before the backslash.

7.2.3 Preventing the make Utility from Echoing Commands

To prevent `make` from echoing the commands that it is executing to standard output, use any one of the following procedures:

- Use the `-s` flag on the command line when you enter the `make` command.
- Put the pseudotarget name `.SILENT:` on a line by itself in the description file. See Section 7.2 for an explanation of pseudotargets.
- Put an at sign (`@`) in the first character position (after the tab) of each command line in the description file that `make` should not echo.

7.2.4 Preventing the make Utility from Stopping on Errors

The `make` utility normally stops if any command returns a nonzero status code to indicate an error.

To prevent `make` from stopping on errors, use any of the following procedures:

- Use the `-i` flag on the command line when you enter the `make` command.
- Put the pseudotarget name `.IGNORE:` on a line by itself in the description file. See Section 7.2 for an explanation of pseudotargets.
- Put a hyphen (`-`) in the first character position (after the tab) of each command line in the description file where `make` should not stop on errors.

7.2.5 Defining Default Conditions

When `make` creates a target but cannot find either explicit command lines or internal rules to create the file, it looks at the description file for default conditions. To define the commands that `make` performs in this case, use the `.DEFAULT:` pseudotarget name in the description file, entering the desired default command sequence as for any other target.

Use the `.DEFAULT:` pseudotarget for an error recovery routine or for a general procedure to create all files in the program that are not defined by an internal rule of the `make` utility.

7.2.6 Preventing make from Deleting Files

To prevent completion of a build using potentially corrupted target files, `make` normally removes target files if an error is returned during the build. To prevent `make` from removing files when an error is detected, use the `.PRECIOUS:` pseudotarget in the description file. After the pseudotarget name, list the target names that should be saved. If you specify the `-u` option on the command line, `make` does not remove any RCS-files it checked out. See the `make`(1) reference page for more information on how `make` interacts with RCS.

7.2.7 Simple Description File

In Example 7-1, a program named `prog` is made by compiling and loading three C language files: `x.c`, `y.c`, and `z.c`. The files `x.c` and `y.c` share some declarations in a file named `defs`. The `z.c` file does not share those declarations.

Example 7-1: A Simple Description File

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
        cc  x.o y.o z.o -o prog

# Make x.o from 2 other files
x.o:   x.c defs
# Use the cc program to make x.o
        cc  -c x.c

# Make y.o from 2 other files
y.o:   y.c defs
# Use the cc program to make y.o
        cc  -c y.c

# Make z.o from z.c
z.o:   z.c
# Use the cc program to make z.o
        cc  -c z.c
```

If this file is called `makefile`, you can enter the `make` command with no options or arguments to make an up-to-date copy of `prog` after making changes to any of the four source files `x.c`, `y.c`, `z.c`, or `defs`.

7.2.8 Making the Description File Simpler

To make the description file simpler, use the internal rules of the `make` utility. Using file system naming conventions, `make` knows that there are three `.c` files corresponding to the needed `.o` files. It also knows how to generate an object from a source file (that is, issue a `cc -c` command). By taking advantage of these internal rules, the description file becomes the following:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog

# Use the file defs and the appropriate .c file
# when making x.o and y.o
x.o y.o:   defs
```

Section 7.2.14 describes the internal rules used by `make`.

7.2.9 Defining Macros

A macro is a name to use in place of one or more other names. It is a shorthand way of using the longer string of characters. You can define macros in the description file or on the command line. To define a macro in the description file, do the following:

1. Start a new line with the name of the macro.
2. Follow the name with an equal sign (=).
3. To the right of the equal sign, enter the string of characters that the macro name represents. The string can contain blanks.

The macro definition can contain blanks before and after the equal sign without affecting the result. The macro definition cannot contain a colon (:) or a tab before the equal sign. The `make` utility ignores leading and trailing blanks in the defining string. The following examples are macro definitions:

```
# Macro ABC has a value of "ls -la"
ABC = ls -la

# Macro LIBES has a null value
LIBES =

# Macro DIRECT includes the definition of macro ROOT
# The expanded value of DIRECT is "/usr/home/fred"
ROOT   = /usr/home
DIRECT = $(ROOT)/fred
```

The `DIRECT` macro in this example uses another definition as part of its own definition. See Section 7.2.10 for instructions on using macros.

To define a macro on a command line, follow the same syntax as for defining macros in the description file, but include all of your macro definitions on the same line. When you define a macro with blanks from the command line, enclose the definition in quotation marks (`"name = definition"`). Without the quotation marks, the shell interprets the blanks as parameter separators and not as part of the macro.

7.2.10 Using Macros in a Description File

After you define a macro in a description file, refer to the macro's value in the description file by putting a dollar sign ($) before the name of the macro. If the macro name is longer than one character, put parentheses or braces around it, as illustrated by the following examples:

```
$(CFLAGS)
${xy}
$Z
$(Z)
```

The effect of the last two examples is identical.

7.2.10.1 Macro Substitution

You can substitute a different value for part or all of a macro's defined value. The three forms of macro substitution are as follows:

- The first form replaces every occurrence of *string1* in the defined value of *MACRO* with *string2*:

  $(*MACRO*:*string1*=*string2*)

  For example:

```
# Define macro MAC1
MAC1 = xxx yyy zzz
   .
   .
   .
# Evaluate MAC1
project:
        @ echo $(MAC1:yyy=abc)
```

  When you run `make` with this description file, `make` substitutes `abc` for the occurrence of `yyy`, and displays the following line:

```
xxx abc zzz
```

- The second form applies a substitution to each word in the defined value. The *location* parameter specifies what portion of the word is to be replaced with *string*:

  $(*MACRO*/*location*/*string*)

  The *location* parameter is restricted to the following values:

  – Circumflex (^) - The *string* value is added as a prefix to each defined word. For example:

```
# Define macro MAC1
MAC1 = abc def ghi
   .
   .
   .
# Evaluate MAC1
project:
        @ echo $(MAC1/^/xyz)
```

    When you run `make` with this description file, `make` adds `xyz` to the beginning of each defined word and displays the following line:

```
xyzabc xyzdef xyzghi
```

  – Asterisk (*) – The *string* value replaces all of each defined word. For example:

```
# Define macro MAC1
MAC1 = abc def ghi
   .
   .
   .
# Evaluate MAC1
project:
        @ echo $(MAC1/*/xyz)
```

When you run `make` with this description file, `make` substitutes `xyz` for each defined word and displays the following line:

```
xyz xyz xyz
```

With the asterisk, you can use an ampersand (`&`) in the *`string`* value. The ampersand represents the defined word that is being substituted for, and it causes that word to be interpolated in the result. For example:

```
# Define macro MAC1
MAC1 = abc def ghi
   •
   •
   •
# Evaluate MAC1
project:
        @ echo $(MAC1/*/x&z)
```

When you run `make` with this description file, `make` substitutes `x&z` for each defined word, interpolating the defined word for the ampersand, and displays the following line:

```
xabcz xdefz xghiz
```

– Dollar sign (`$`) - The *`string`* value is appended to each defined word. For example:

```
# Define macro MAC1
MAC1 = abc def ghi
   •
   •
   •
# Evaluate MAC1
project:
        @ echo $(MAC1/$/xyz)
```

When you run `make` with this description file, `make` appends `xyz` to the end of each defined word and displays the following line:

```
abcxyz defxyz ghixyz
```

- The third form makes one of two possible substitutions depending on whether *MACRO* is defined:

  $(*MACRO***?***string1***:***string2***)**

  If *MACRO* is defined, *`string1`* is substituted for the entire defined value. If *MACRO* is not defined, *`string2`* is used. For example:

```
# Define macro MAC1
MAC1 = abc def ghi
   •
   •
   •
# Evaluate MAC1 and MAC2.  MAC2 is not defined.
project:
        @ echo $(MAC1?uvw:xyz)
        @ echo $(MAC2?123:456)
```

When you run make with this description file, make substitutes uvw for the value of MAC1 and 456 for the undefined MAC2, and displays the following lines:

```
uvw
456
```

The first two forms of substitution produce a null string if *MACRO* is undefined.

7.2.10.2 Conditional Macros

The value of a macro can be assigned based on a preexisting condition. This type of macro is a conditional macro. You cannot define conditional macros on the command line; all conditional macro definitions must be in the description file. The syntax of the conditional macro is as follows:

target:=MACRO=string

The macro is assigned the value of the string if the specified target is the current target of the make command. Otherwise, the macro's value is null. The following description files uses a conditional substitution for MAC1:

```
# Define the conditional macro MAC1
target2:=MAC1 = xxx yyy xxxyyy
   •
   •
   •
#list targets and command lines
#
target1:;@echo $(MAC1)
target2:;@echo $(MAC1)
```

When you run make with this description file, you get the following results:

```
% make target1

% make target2
xxx yyy xxxyyy
```

7.2.11 Calling the make Utility from a Description File

You can nest calls to the make utility within a make description file by including the $(MAKE) NX r "make macro=>$(MAKE) macro" macro in one of the command lines in the file. If this macro is present, make executes another copy of make, even if the -n option is set. See Section 7.2.16 for a description of the -n option.

7.2.12 Internal Macros

The make utility has built-in macro definitions for use in the description file. These macros help specify variables in the description file. The make utility replaces the macros with the values indicated in Table 7-1.

Table 7-1: Internal make Macros

Macro	Value
$@	The name of the current target file
$$@	The target names on the dependency line
$?	The names of the dependency files that have changed more recently than the target
$<	The name of the out-of-date file that caused a target file to be created
$*	The name of the current dependency file without the suffix

Each of these macros resolves to a single file name at the time make is actually using it. You can modify the interpretation of any of these macros by using a D suffix to indicate that you want only the directory portion of the name. For example, if the current target is /u/tom/bin/fred, the $(@D) macro returns only the /u/tom/bin portion of the name. Similarly, an F suffix returns only the file name portion. For example, the $(@F) macro returns fred if given the same target. Note the following exception: All internal macros except the $? macro can take the D or F suffix.

Before using any internal macros on a distributed file system, you must ensure that the system clocks show the same date and time for all nodes that contain files for make to process.

The make utility replaces these symbols only when it runs commands from the description file to create the target file. The following sections explain these macros in more detail.

7.2.12.1 Internal Target File Name Macro

The make utility substitutes the full name of the current target for every occurrence of the $@ macro in the command sequence for building the target. The replacement is made before running the command. For example:

```
/u/tom/bin/test:  test.o
        cc test.o -o $@
```

This example produces an executable file named /u/tom/bin/test.

7.2.12.2 Internal Label Name Macro

If the $$@ macro is used on the right side of the colon on a dependency line in a description file, make replaces this symbol with the label name that is on the left side of the colon in the dependency line. This name could be a target name or the name of another macro. For example:

```
cat:  $$@.c
```

The make utility interprets this line as follows:

```
cat:  cat.c
```

Use this macro to build a group of files, each of which has only one source file. For example, to maintain a directory of system commands, use a description file like the following:

```
# Define macro CMDS as a series of command names
CMDS = cat dd echo date cc cmp comm ar ld chown

# Each command depends on a .c file
$(CMDS):  $$@.c

# Create the new command set by compiling the out of
# date files ($?) to the current target file name ($@)
        cc -O $? -o $@
```

The make utility changes the $$(@F) macro to the file part of $@ when it runs. For example, you could use this symbol when maintaining the usr/include directory while using a description file in another directory. That description file would look like the following example:

```
# Define directory name macro INCDIR
INCDIR = /usr/include

# Define a group of files in the directory
# with the macro name INCLUDES
INCLUDES = \
         $(INCDIR)/stdio.h \
         $(INCDIR)/pwd.h \
         $(INCDIR)/dir.h \
         $(INCDIR)/a.out.h

# Each file in the list depends on a file
```

```
# of the same name in the current directory
$(INCLUDES):       $$(@F)

# Copy the younger files from the current
# directory to /usr/include
         cp $? $@

# Set the target files to read only status
         chmod 0444 $@
```

This description file creates a file in the `/usr/include` directory when the corresponding file in the current directory has been changed.

7.2.12.3 Internal Younger Files Macro

If the `$?` macro is in the command sequence in the description file, `make` replaces the symbol with a list of dependency files that have been changed since the target file was last changed.

7.2.12.4 Internal First Out-of-Date File Macro

If the `$<` macro is in the command sequence in the description file, `make` replaces the symbol with the name of the file that started the file creation. The file name is the name of the specific dependency file that was out of date with the target file and therefore caused `make` to create the target file again. Note the difference between this symbol and the `$?` symbol, which returns a complete list of younger files.

Note that the `make` utility replaces this symbol only when it runs commands from its internal rules or from the `.DEFAULT:` list. The symbol has no effect in an explicitly stated command line.

7.2.12.5 Internal Current File Name Prefix Macro

If the `$*` macro is in the command sequence in the description file, `make` replaces the symbol with the file name part (without the suffix) of the dependency file that `make` is currently using to generate the target file. For example, if `make` is building the target `test.c`, the `$*` symbol represents the file name `test`.

The `make` utility replaces this symbol only when it runs commands from its internal rules or from the `.DEFAULT:` list. The symbol has no effect in an explicitly stated command line.

7.2.13 How make Uses Environment Variables

Each time `make` runs, it reads the current environment variables and adds them to its defined macros. In addition, it creates a new macro called `MAKEFLAGS`. This macro is a collection of all the options that were entered

on the command line. Command-line options and assignments in the description file can also change the value of the MAKEFLAGS macro. When make starts another process, it exports MAKEFLAGS to that process. See Section 7.2.16 for a discussion of how the MAKEFLAGS macro affects recursive make processes.

Macro definitions are assigned in the following order when make:

1. Reads the MAKEFLAGS environment variable to set options specified by the variable. If MAKEFLAGS is not present or is null, make sets its internal MAKEFLAGS macro to the null string. Otherwise, make assumes that each letter in MAKEFLAGS is an input option. The make utility uses these options (except for –f, –p, and –r) to determine its operating conditions.
2. Reads and sets the input flags from the command line. Any options specified explicitly on the command line are added to the settings from the MAKEFLAGS environment variable.
3. Reads macro definitions from the command line. These definitions override any definitions for the same names in the description file.
4. Reads the internal macro definitions.
5. Reads the environment, including the MAKEFLAGS macro. The make utility treats all environment variables as macro definitions and passes them to shells it invokes to execute commands.

7.2.14 Internal Rules

The make utility has a set of internal rules that it uses to determine how to build a target. You can override these rules by invoking make with the -r option; in this case, you must supply any rules that are required to build the targets in your description file. The internal rules contain a list of file name suffixes defined using the pseudotarget .SUFFIXES:, along with the rules that tell make how to create a file with one suffix from a file with another suffix. To see the complete list of conversions supported by make's internal rules, run the following command:

```
% make -p | more
```

If you do not change the list, make by default understands the following suffixes:

Suffix	File Type
.o	Object file
.c	C source file
.e	efl source file

Suffix	File Type
.r	Ratfor source file
.f	FORTRAN source file
.s	Assembler source file
.y	yacc C source grammar
.yr	yacc Ratfor source grammar
.ye	yacc efl source grammar
.l	lex source grammar
.out	Executable file
.F	FORTRAN source file
.p	Pascal source file
.sh	Bourne shell script
.csh	C shell script
.h	C header file

You can add suffixes to this list by including a `.SUFFIXES:` line in the description file with one or more space-separated suffixes. For example, the following line adds the suffixes `.f77` and `.ksh` to the existing list. For example:

```
.SUFFIXES: .f77 .ksh
```

To erase `make`'s default list of suffixes, include a `.SUFFIXES:` line with no names on it. You can replace the default list with a completely new list by using first an empty list and then your new list:

```
.SUFFIXES:
.SUFFIXES: .o .c .p .sh .ksh .csh
```

Because `make` looks at the suffixes list in left-to-right order, the order of the entries is important. The preceding example ensures that `make` will look first for an object file, then a C source file, and so on.

The `make` utility uses the first entry in the list that satisfies the following two requirements:

- The entry matches input and output suffix requirements.
- The entry has a rule assigned to it.

If you add suffixes to the list that `make` recognizes, you must provide rules that describe how to build a target from its dependents. A rule looks like a dependency line and the corresponding series of commands. The `make` utility creates the name of the rule from the two suffixes of the files that the

rule defines. For example, the name of the rule to transform a `.r` file to a `.o` file is `.r.o`. Example 7-2 illustrates a portion of the standard default rules file.

Example 7-2: Default Rules File

```
# Create a .o file from a .c
# file with the cc program
.c.o
        $(CC) $(CFLAGS) -c $<

# Create a .o file from either a
# .e , a .r , or a .f
# file with the efl compiler
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<

# Create a .o file from
# a .s file with the assembler
.s.o:

       $(AS) -o $@ $<

.y.o:
# Use yacc to create an intermediate file

       $(YACC) $(YFLAGS) $<
# Use cc compiler

       $(CC) $(CFLAGS) -c y.tab.c
# Erase the intermediate file

       rm y.tab.c
# Move to target file

       mv y.tab.o $@

.y.c:
# Use yacc to create an intermediate file

       $(YACC) $(YFLAGS) $<
# Move to target file

       mv y.tab.c $@
```

7.2.14.1 Single Suffix Rules

The `make` utility also has a set of single suffix rules to create targets with no suffixes, such as command files. The `make` utility has rules to change the following source files with a suffix to object files without a suffix:

Suffix	Source File Type
.c	From a C language source file
.sh	From a shell file

For example, to maintain a program like `cat` if all of the needed files are in the current directory, enter the following command:

```
% make cat
```

7.2.14.2 Overriding Built-In make Macros

The `make` utility uses macro definitions in its internal rules. To change these macro definitions, enter new definitions for those macros on the command line or in the description file. For commands and language processors, the `make` utility uses the following macro names:

Command or Function	Command Macro	Command Options or Other Macros
Archive program (`ar`)	AR	ARFLAGS
Archive table of contents creation		RANLIB
Assembler	AS	ASFLAGS
C Compiler	CC	CFLAGS
C libraries		LOADLIBS
RCS checkout	CO	COFLAGS
The copy command (`cp`)	CP	CPFLAGS
`efl` compiler	EC	EFLAGS
Linker command (`ld`)	LD	LDFLAGS
The `lex` command	LEX	LFLAGS
The `lint` command	LINT	LINTFLAGS
The `make` command	MAKE	
Recursive `make` calling flags		MAKEFLAGS
The `mv` command	MV	MVFLAGS
The `pc` command	PC	PFLAGS
The `f77` compiler	RC	FFLAGS
Ratfor compiler flags		RFLAGS
The `rm` command	RM	RMFLAGS
For locating files related to dependency		VPATH

Command or Function	Command Macro	Command Options or Other Macros
The `yacc` command	YACC	YFLAGS
The `yacc -e` command	YACCE	YFLAGS
The `yacc -r` command	YACCR	YFLAGS

For example, the following command runs `make`, substituting the `newcc` program in place of the previously defined C language compiler:

```
% make CC=newcc
```

Similarly, the following command tells `make` to optimize the final object code produced by the C language compiler.

```
% make "CFLAGS=-O"
```

To look at the internal rules that `make` uses, enter the following command from the Bourne shell:

```
$ make -fp -< /dev/null 2>/dev/null
```

The output appears on the standard output.

7.2.15 Including Other Files

You can include files in addition to the current description file by using the word `include` as the first word on any line in the description file. Follow the word with a blank or a tab and then the set of file names for `make` to include in the operation. For example:

```
include    /u/tom/temp /u/tom/sample
```

7.2.16 Testing Description Files

To test a description file, run `make` with the `-n` command option. This option instructs `make` to echo command lines without executing them. Even commands preceded by at signs (`@`) are echoed so that you can see the entire process as `make` would execute it. When the `-n` option is in effect, the NX r "make macro=>$(MAKE) macro" "testing description files with" `$(MAKE)` macro, unlike all other commands, is actually executed. If the description file includes an instance of the `$(MAKE)` macro, `make` calls the new copy of `make` with the `MAKEFLAGS` macro's value set to the list of options, including `-n`, that you entered on the command line. The new copy of `make` observes that the `-n` option is set, and it bypasses command execution in the

same way as the copy that called it. You can test a set of description files that use recursive calls to `make` by entering a single `make` command.

7.2.17 Description File

Example 7-3 shows the description file that maintains the `make` utility. The source code for `make` is contained in a number of C language source files and a `yacc` grammar file. For more information on `yacc`, see Chapter 4.

Example 7-3: The makefile for the make Utility

```
# Description file for the Make program

# Macro def: send to be printed
P = lpr

# Macro def: source file names used
FILES = Makefile version.c defs main.c

      doname.c misc.c files.c
      dosy.c gram.y lex.c gcos.c

# Macro def: object file names used
OBJECTS = version.o main.o doname.o \
         misc.o files.o dosys.o gram.o

# Macro def: lint program and flags
LINT = lint -p

# Macro def: C compiler flags
CFLAGS = -O

# make depends on the files specified
# in the OBJECTS macro definition
make:    $(OBJECTS)
# Build make with the cc program
        cc $(CFLAGS) $(OBJECTS) -o make
# Show the file sizes
        size make

# The object files depend on a file
# named defs
$(OBJECTS):  defs

# The file gram.o depends on lex.c
# uses internal rules to build gram.o
gram.o:  lex.c

# Clean up the intermediate files
clean:
         rm *.o gram.c
```

Example 7-3: (continued)

```
# Copy the newly created program
# to /usr/bin and deletes the program
# from the current directory
install:
        cp make /usr/bin/make ; rm make

# Empty file ''print'' depends on the
# files included in the macro FILES
print:   $(FILES)
# Print the recently changed files
        lpr $?
# Change the date on the empty file,
# print, to show the date of the last
# printing
        touch print

# Check the date of the old
# file against the date
# of the newly created file
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

# The program, lint, depends on the
# files that are listed
lint:    dosys.c doname.c files.c main.c misc.c \
        version.c gram.c
# Run lint on the files listed
# LINT is an internal macro
        $(LINT) dosys. doname.c files.c main.c \
        misc.c version.c gram.c
        rm gram.c

# Archive the files that build make
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Creating and Managing Software Product Kits 8

Data transfer is the process of creating at a target location an exact copy of information that is at some other location (the source). The directories and files that make up a software product kit can be viewed as a set of data. Digital's `setld` architecture is a mechanism for data transfer, by which a copy of a software kit (the data) is transferred from a vendor's system to a customer's system. This chapter describes Digital's `setld` utility. After a brief description of how you use `setld` to install, manage, and remove software kits, the major part of the chapter describes how to create kits for `setld`. All DEC OSF/1 software kits supplied by Digital are `setld`-compatible, and many other software vendors also supply their products in this form. Providing your software as a `setld`-compatible kit has the following benefits:

- Installation security

  When you use the `setld` utility to install a software product, each installable component (called a **subset**) of the product is verified immediately after transfer. Each subset is recoverable if you want to reinstall it or if it is damaged or deleted.

- Flexibility

  The `setld` utility allows you to choose subsets at installation and to delete subsets. These facilities simplify the process of customizing a system for specific types of activities. For example, after installing the mandatory system subsets, you might load optional subsets and products for communications, document creation, database operations, or other purposes.

- Uniformity

  The `setld` utility is an integral part of Digital's installation architecture. Producing products in the form of `setld`-compatible kits enhances your products' compatibility with products produced by Digital and other software vendors.

- Media support

  You can use any of the following devices to install a `setld`-compatible software product from the distribution media specified:

  – An arbitrary, mountable file system on any supported data disk; for example, a third-party SCSI disk cartridge.

- A TK50 cartridge tape on a TK50 or a TZ30 tape drive
- A 9-track (MT9) tape of arbitrary density

Digital provides the tools for creating `setld` kits as part of the standard operating system distribution; these tools are described in Section 8.5.

Examples in this chapter describe a hypothetical kit for a product called "Orpheus Authoring Tools." The example kit is considered to be one of several Orpheus products. Because this particular kit is a document builder, the kit name is abbreviated as DCB and the main project directory is `dcb_tools`.

8.1 The setld Command and Its Functions

The `setld` utility is an interactive program for managing software subsets. A **subset** is the smallest installable entity in a software kit; it can contain any mixture of source, binary, and documentation files. A kit can contain any number of subsets.

The syntax of the `setld` utility is as follows:

setld [**-D** *root-path*] **-c** *subset-id message*
setld [**-D** *root-path*] **-d** *subset-id* [*subset-id...*]
setld [**-D** *root-path*] **-i** [*subset-id* [*subset-id...*]]
setld [**-D** *root-path*] **-l** *location* [*subset-id* [*subset-id...*]]
setld [**-D** *root-path*] **-v** *subset-id* [*subset-id...*]

Table 8-1 provides brief descriptions of the `setld` utility's options. See the *System Administration* for more thorough instructions on using `setld`.

Table 8-1: Options for the setld Command

Option	Description

`-D`	In conjunction with any other option, specifies an alternative root directory. For example:
	`# setld -D /usr/doctools -i`
	Note that the –D option causes `setld` to bypass the `C INSTALL` phase of an installation. To force the `C INSTALL` phase to execute, specify the –c option together with the –D option. For example:
	`# setld -D /usr/doctools -c OATDCB100 INSTALL`

Table 8-1: (continued)

Option	Description
-l	Commands `setld` to install the software kits that are in the specified location; if subsets are specified, only the named subsets are installed. For example: `# setld -l /dev/rmt0h OATDCB100`
-c	Commands `setld` to run the named subset's subset control program (SCP) to execute its configuration code. This option passes a message to the SCP; currently, the only supported standard messages are `INSTALL` (to run the `C INSTALL` code) and `DELETE` (to run the `C DELETE` code). For example: `# setld -c OATDCB100 INSTALL` You can use the –c option to pass nonstandard messages to a subset, but such messages will have an effect only if the target subset's SCP is coded to accept them.
-i	Displays a list of subsets and their installation status; or, if a subset is named, displays a list of that subset's contents. For example: `# setld -i OATDCB100`
-v	Commands `setld` to run the named subset's SCP to execute the subset's Installation Verification Procedure (IVP). For example: `# setld -v OATDCB100`
-d	Commands `setld` to delete the named subset or subsets. For example: `# setld -d OATDCB100 OATDCBDOC100`

8.2 Files Used by the setld Utility

In addition to the subset files themselves, `setld` uses several other files to install a product kit. When a kit is built, a file named `INSTCTRL` is included. This file is a `tar` archive of the following files:

- The image data file, *`product-code`*`.image`
- The compression flag file, *`product-id`*`.comp`
- A subset control file, *`subset-id`*`.ctrl`, for each subset in the kit
- A subset inventory file, *`subset-id`*`.inv`, for each subset in the kit
- A subset control program (SCP), *`subset-id`*`.scp`, for each subset in the kit

Except for the SCPs, these files are constructed by the `kits` command (described in Section 8.5.4) from information in the product's master inventory and key files, described in Section 8.5.2, and from information compiled during the actual kit building process (such as subset sizes and checksums). During product installation procedure, these files are placed in the `/usr/.smdb.` directory, where they will be available for reference and reuse by later invocations of `setld`. Refer to Section 8.5.4 for descriptions of the various files used except for the SCPs, which are described in Section 8.5.3.

In addition to the files extracted from kits, `setld` also creates and modifies lock files. Each subset in the system's inventory has a lock file. Lock files are of the following two types:

- A lock file indicating successful installation of a subset, named *subset-id*`.lk`
- A lock file indicating failed (''corrupt'') installation of a subset, named *subset-id*`.dw`

When a subset is installed, one of these two lock files is created. At that time, the lock file is empty. Assuming successful installation, that subset is then available for dependency checks and locking performed on behalf of subsets installed later.

For example, the DCB kit requires that some version of the Orpheus Authoring Tools base (a different product) be installed in order for the DCB product to work properly. Suppose that the OATBASE200 subset is present. When `setld` installs the OATDCB100 subset from the DCB kit, it inserts a record containing the subset identifier OATDCB100 into the `OATBASE200.lk` file. (A given subset's lock file can contain any number of records, each naming a single dependent subset.) When the system administrator uses `setld` to remove the OATBASE200 subset, `setld` checks `OATBASE200.lk` and finds a record indicating that OATDCB100 depends on OATBASE200; `setld` displays a warning message with this information and requires confirmation that the user really intends to remove the OATBASE200 subset.

If the administrator removes the OATDCB100 subset, `setld` removes the corresponding record from the `OATBASE200.lk` file; thereafter, OATBASE200 can be removed without flagging a dependency warning.

8.3 Descriptions of setld Functions

The following sections describe the steps the `setld` utility performs in response to an invocation with each of its options.

Note

The `setld` command's action is divided into phases. Some phases have `PRE_`*`phase`* and `POST_`*`phase`* subphases. If a given subset's `PRE_`*`phase`* subphase fails during any applicable operation, `setld` displays a message indicating that the SCP has declined the operation and does not proceed further with that subset. No attempt is made to run the *`phase`* or `POST_`*`phase`* code.

8.3.1 Loading Software

When you load software by using the `–l` option to `setld`, the utility performs the following steps:

1. Verifies access to *`location`*.
2. Copies product installation information from *`location`* into a temporary area. The information copied is contained in the `INSTCTRL` file for each product kit to be installed. If the `setld` command line included specific subset identifiers, only those subsets are considered; otherwise, all subsets in *`location`* are considered.
3. Determines which subsets to load by calling each subset's SCP with the `ACT` environment variable set to `M`. Subsets whose SCPs determine that their respective subsets are candidates for installation are divided into mandatory and optional groups according to the subset control flags contained in the *`subset-id`*`.ctrl` files.
4. Displays a list of the candidate subsets, listing the mandatory subsets (if any) and offering the optional subsets in a menu for selection by the user. If there are no optional subsets, no menu is displayed; instead, the mandatory subsets are listed and the user is asked for permission to proceed.
5. Performs the following operations for each subset to be installed:
   a. Creates a ''subset corrupt'' lock file.
   b. Verifies that the subset will fit on the system.
   c. Invokes the subset's SCP to perform product-specific tasks that must be done before the subset is loaded (`ACT` set to `PRE_L`). A nonzero return status from the SCP will cause `setld` to abort the load operation.
   d. Loads the subset, using the subset's control and inventory files; then verifies the subset and upgrades the lock file to indicate that the subset is correctly installed.

6. After loading all the chosen subsets, performs the following steps for each subset:

   a. Invokes the subset's SCP to perform product-specific tasks that must be done after the subset is loaded (`ACT` set to `POST_L`). The SCP's actions at this time usually include dependency locking.

   b. Invokes the SCP to perform product-specific tasks that must be done after the subset is installed (`ACT` set to `C` and `$1` set to `INSTALL`). This step is bypassed if the `–D` command option was invoked.

The installation control files (*`subset-id`*`.ctrl`, *`subset-id`*`.inv`, *`subset-id`*`.scp`, and *`subset-id`*`.lk` or *`subset-id`*`.dw`) are stored in the `./usr/.smdb.` directory. The kit's subset archives themselves are not stored, since their contents have been placed in the appropriate locations. If you specified an alternative root path, this directory path is created under the directory you specify.

8.3.2 Configuring a Subset

When you load a product, the next-to-last stage of the `setld` process is to invoke the subset's SCP with the `ACT` environment variable set to `C` and the command argument (`$1`) set to `INSTALL`. See Section 8.5.3 for a description of how the SCP responds.

When you issue a command to reconfigure a subset (the `–c` option), `setld` first verifies that the specified subset exists. If it does, `setld` sets the `ACT` environment variable to `C` and calls the subset's SCP with *`message`* as a command argument (`$1`). Normally, the only valid messages are `INSTALL` and `DELETE`. These two messages are reserved in their meaning; see Table 8-7. For special needs, a particular SCP could be designed to accept other messages. The `setld` utility cannot pass such other messages except in response to its `–c` option.

8.3.3 Verifying a Subset

When you load a product, the final stage of the `setld` process is to invoke the subset's SCP with the `ACT` environment variable set to `V`. This action instructs the SCP to run its verification test. See Section 8.5.3 for a description of how the SCP responds.

When you issue a command to reverify a subset (the `–v` option), `setld` first verifies that the specified subset exists. If it does, `setld` runs the subset's Installation Verification Procedure (IVP), if there is one.

8.3.4 Removing Software

When you issue a `setld -d` command, the `setld` utility performs the following steps for each subset to be deleted:

1. Verifies that the subset is installed.
2. Verifies that the subset's "sticky" bit, originally specified in the product's key file, is not set. If the "sticky" bit is set, `setld` declines to remove the subset.
3. Checks dependencies. If the subset's lock file (*`subset-id`*`.lk`) names any subsets that depend on the one to be removed, `setld` displays their names and requests confirmation that the subset should be deleted.
4. Invokes the subset's SCP to perform product-specific tasks that must be done before any deletions are made. (`ACT` set to `C` and `$1` set to `DELETE`.)
5. Invokes the subset's SCP to perform product-specific tasks that must be done before the subset is deleted. (`ACT` set to `PRE_D`.) If the SCP returns nonzero status, `setld` aborts the deletion operation.
6. Deletes all files contained in the subset.
7. Invokes the subset's SCP to perform product-specific tasks that must be done after the subset is deleted. (`ACT` set to `POST_D`.)
8. Marks the subset as being uninstalled by deleting its lock file.

8.4 Using the File System Effectively

The `setld` utility allows the software vendor to install the components of a kit in any directories desired. However, guidelines exist for deciding where to place kit elements. This section describes how to design your kit's layout to make proper and effective use of the file system.

8.4.1 Using Standard Directories

The standard DEC OSF/1 system directory hierarchy is set up for efficient organization. It separates files by function and intended use. Effective use of the file system includes placing command files in directories that are in the normal search path as specified by users' `.profile` or `.login` files, as appropriate. Note, however, that the physical location of layered products is governed by a special set of guidelines described in Section 8.4.2. The important directories in the file system are shown in Figure 8-1 and Figure 8-2. Not all of the directories in the entire hierarchy are shown; the directories that are illustrated are the ones that you should use to ensure that your product will be portable to other systems. Some of the illustrated directories are actually symbolic links.

Figure 8-1: Base System Directory Hierarchy

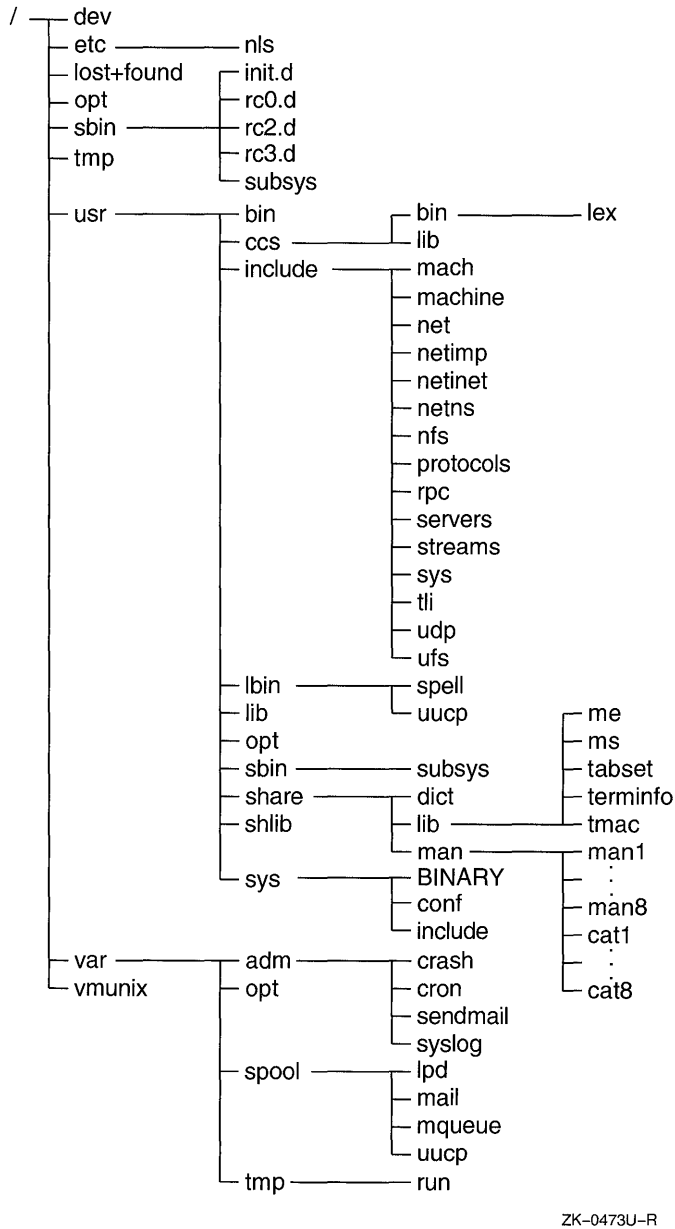


Table 8-2 describes the contents and purposes of the directories shown in Figure 8-1.

Table 8-2: Contents and Purposes of Base System Directories

Directory	Description
/	The root directory of the file system
/dev/	Block and character device files
/etc/	System configuration files and databases; nonexecutable files
nls/	National language support databases
/lost+found/	Files located by `fsck`
/opt/	Optional application packages such as layered products
/sbin/	Commands essential to boot the system
	These commands do not depend on shared libraries or the loader and can have other versions in `/usr/bin` or `/usr/sbin`.
init.d/	System state rc files
rc0.d/	The rc files executed for system-state 0
rc2.d/	The rc files executed for system-state 2
rc3.d/	The rc files executed for system-state 3
subsys/	Loadable kernel modules required in single-user mode
/tmp/	System-generated temporary files
	The contents of `/tmp` are usually not preserved across a system reboot.
/usr/	Most user utilities and applications
bin/	Common utilities and applications
ccs/	C compilation system; tools and libraries used to generate C programs
bin/	Development binaries such as `cc`, `ld`, and `make`
lib/	Development libraries and back ends
lex/	Data for `lex`
include/	Program header (include) files; not all subdirectories are listed below
mach/	Mach-specific C include files
machine/	Machine-specific C include files
net/	Miscellaneous network C include files
netimp/	C include files for IMP protocols
netinet/	C include files for Internet standard protocols
netns/	C include files for XNS standard protocols

Table 8-2: (continued)

Directory	Description
nfs/	C include files for Network File System
protocols/	C include files for Berkeley service protocols
rpc/	C include files for remote procedure calls
servers/	C include files for servers
streams/	C include files for STREAMS
sys/	System C include files (kernel data structures)
tli/	C include files for Transport Layer Interface
udp/	C include files for User Datagram Protocol
ufs/	C include files for UNIX File System
lbin/	Back-end executables
spell/	Spell back-end
uucp/	UNIX-to-UNIX Copy (UUCP) programs
lib/	Consists entirely of links to libraries located elsewhere (/usr/ccs/lib), (/usr/libin), (/usr/share/lib), (/X11/lib); included for compatibility
opt/	Optional application packages such as layered products
sbin/	System administration utilities and system utilities
subsys/	Loadable kernel modules required in single-user mode
share/	Architecture-independent ASCII text files
dict/	Word lists
lib/	Various libraries
me/	Macros for use with the me macro package
ms/	Macros for use with the ms macro package
tabset/	Tab description files for a variety of terminals; used in /etc/termcap
terminfo/	Terminal information database
tmac/	Text processing macros
man/	Online reference pages
man1/	Source for user command reference pages
man2/	Source for system call reference pages
man3/	Source for library routine reference pages
man4/	Source for file format reference pages
man5/	Source for miscellaneous reference pages
man7/	Source for device reference pages

Table 8-2: (continued)

Directory	Description
man8/	Source for administrator command reference pages
cat1-cat8	Formatted versions of reference pages in the man1 through man8 directories
shlib/	Binary loadable shared libraries; shared versions of libraries in /usr/ccs/lib
sys/	System configuration files
BINARY	Object files
conf/	Kernel configuration control files
include/	Header files
/var/	Multipurpose log, temporary, transient, varying, and spool files
adm/	Common administrative files and databases
crash/	For saving kernel crash dumps
cron/	Files used by cron
sendmail/	Configuration and database files for sendmail
syslog/	Files generated by syslog
opt/	Optional application packages such as layered products
spool/	Miscellaneous printer and mail system spooling directories
lpd/	Line printer spooling directories
mail/	Incoming mail messages
mqueue/	Undelivered mail queue
uucp/	UUCP spool directory
tmp/	Application-generated temporary files that are kept between system reboots
run/	Files created when daemons are running
/vmunix	Pure kernel executable (the operating system loaded into memory at boot time)

Figure 8-2: X Directory Hierarchy

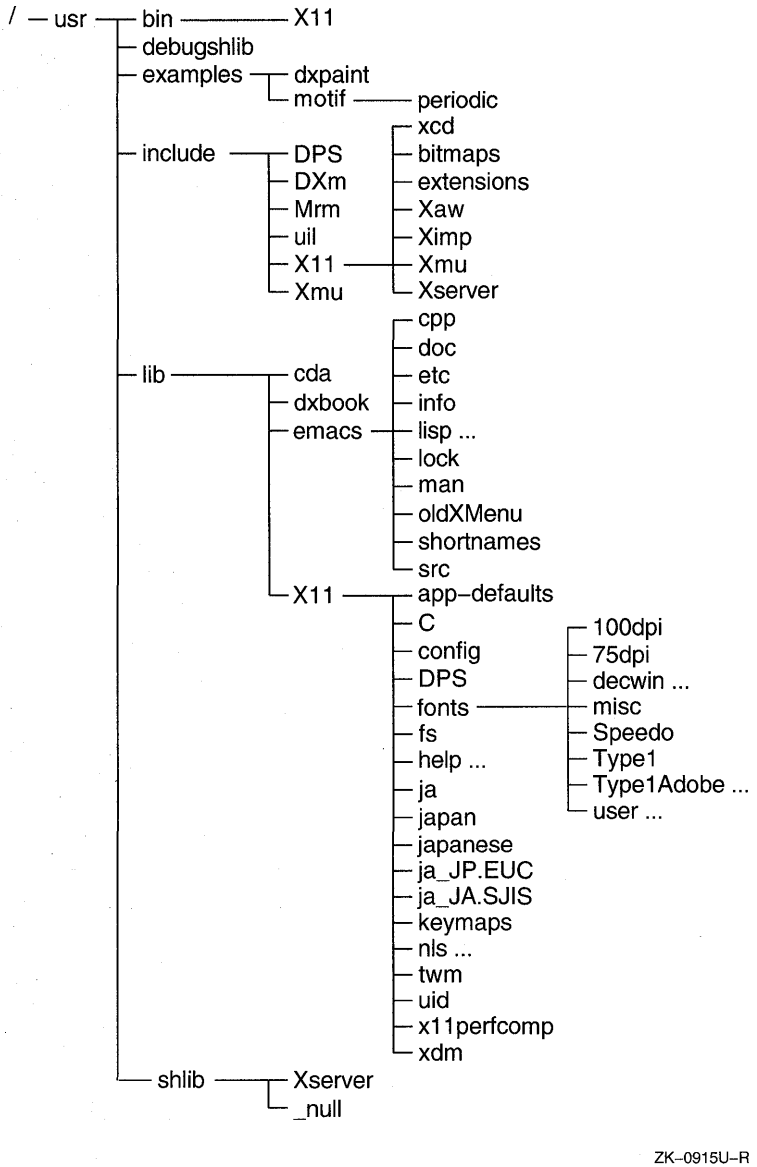


Table 8-3 describes the contents and purposes of the directories shown in Figure 8-2.

Table 8-3: Contents and Purposes of X Directories

Directory	Description
/usr/	Most user utilities and applications
bin/	Common utilities and applications
X11/	X applications
debugshlib/	X shareable libraries compiled with debug information
examples/	Example programs
dxpaint/	Sample Paint image
motif/	Motif example programs
periodic/	Motif periodic widget table example
xcd/	Motif CD player example program
include/	Header files
DPS/	Files for DPS
DXm/	Files for libDXm
Mrm/	Files for libMrm
X11/	X C header files
Xaw/	Files for libXaw
Ximp/	Files for libXimp
Xmu/	Files for libXmu
Xserver/	Header files used for loadable X server libraries
bitmaps/	X bitmaps
extensions/	Header files for use with X extensions
Xm/	Header files for libXm
uil/	UIL header files
lib/	Static archive X libraries
X11	
C/	Internationalization
DPS/	Display Postscript files
app-defaults/	System-wide resource files for X client applications
config/	Imake config files
fonts/	Font files
100dpi/	100 dpi fonts from X Consortium
75dpi/	75 dpi fonts from X Consortium

Table 8-3: (continued)

Directory	Description
decwin/	DECwindows fonts
100dpi/	100 dpi fonts
75dpi/	75 dpi fonts
misc/	Fonts from X Consortium
Speedo/	Speedo scalable fonts
Type1/	Type1 scalable fonts
Type1Adobe/	Adobe Type1 scalable fonts
afm/	Adobe font metrics
user	Fonts from layered products and local installations
100dpi/	100 dpi fonts
75dpi/	75 dpi fonts
misc/	Other fonts
fs/	Fontserver config and error log files
help/	Help files for X client applications; subdirectories as applicable
ja/	Internationalization
ja_JP.EUC/	Internationalization
ja_JP.SJIS/	Internationalization
japan/	Internationalization
japanese/	Internationalization
keymaps/	keymaps for various keyboards
nls/	Internationalization
local_im_tbl/	Internationalization
twm/	Default configuration for `twm` window manager
uid/	User Interface Definitions for X client applications
x11perfcomp/	Scripts for analyzing `x11perf` output
xdm/	X Display Manager configuration and resource files, and error log
cda/	CDA style guides
dxbook/	Default Bookreader bookshelf
emacs/	Emacs directory base
cpp/	Old public cpp

Table 8-3: (continued)

Directory	Description
doc/	PostScript documentation
etc/	Miscellaneous Emacs commands and documentation
info/	Textinfo files
lisp/	LISP source and compiled LISP files
lisp/term/	LISP source and compiled LISP files for term
lock/	Directory to hold file locks
man/	Manual files
oldXMenu/	Sources
shortnames/	Part of source
src/	Sources
shlib/	Shareable libraries
Xserver/	Shareable libraries loaded by X server
_null/	Older versions of shareable libraries

8.4.2 Placing Layered Product Files

An optional, or **layered,** product should be designed so that the user sees it as an integral part of the system. This means that, as with base-system software, layered-product programs such as commands and utilities should be placed in directories that are part of the normal search path, as described in Section 8.4.1. Similarly, libraries should be placed in directories where users would expect to find them. The example DCB kit places command files in a standard system directory (`/usr/bin`), the product's documentation in a directory created by another layered product (`/usr/lib/br`), and template files for users employing the product in a directory unique to the DCB product (`/usr/lib/doclib/templates`).

None of the files for the DCB kit, however, is physically located in any of the directories listed in the preceding paragraph. The DEC OSF/1 system provides special locations (the `opt` directories) for optional products. Files are installed in directories under `/opt`, `/usr/opt`, and `/var/opt` so that the files are centrally located and easy to find. Then a symbolic link is created for each file that makes the file accessible through the traditional directories. For example, the DCB kit's `/usr/bin/attr` command is a link to `/usr/opt/OAT100/bin/attr`. See Example 8-4 for an illustration of the correct way to create these links.

Any given product consists of one or more of the following:

- Files that are nominally read-only; for example, commands, startup files (which can be modified but not by individual users), or prototype data files. These files are installed under `/usr/opt`. For example:

  ```
  /usr/opt/OAT100/bin/attr
  ```

- Files that can be read and written by users; for example, lists of employee telephone numbers. These files are installed under `/var/opt`. For example:

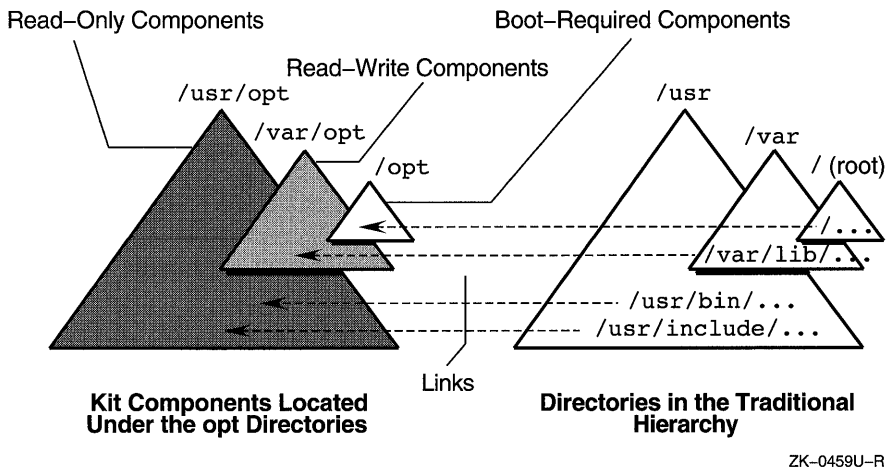
  ```
  /var/opt/UOU100/etc/phonelist
  ```

- Files that are required at boot time; for example, file system drivers. These files are placed under the `/opt` directory. For example:

  ```
  /opt/UOU100/sampledriver
  ```

Figure 8-3 illustrates how a kit is installed and made available in the traditional directories.

Figure 8-3: How Layered Products Are Installed



Digital recommends that you design your product to be installed in the `opt` directories as illustrated. This architecture gives you the following advantages:

- Standardized product design and location.
- If disk partition restructuring or product maintenance becomes necessary, it is much easier to find all of your kit than if its components were scattered throughout the traditional directories.

- Exporting software to share across a network is simplified and more secure; you need only export the specific directories under `/opt`, `/usr/opt`, and `/var/opt` that contain the desired product and then create links on the importing system.

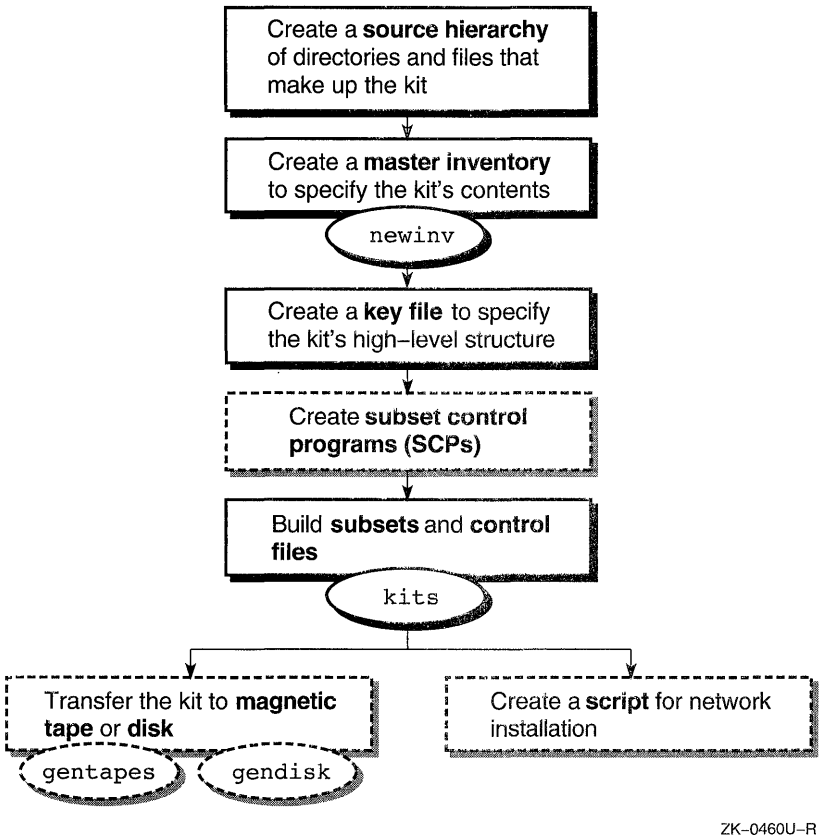
  You can set up a server with multiple versions of a given product, using the links created on the clients to determine which version a given client uses. This architecture also permits maintaining software for multiple dissimilar hardware platforms on the same server. Section 8.5.1 explains the directory naming conventions Digital recommends for kits conforming to this architecture.

The example DCB kit used throughout this chapter adheres to the guidelines described here.

8.5 Creating Kits for the setld Utility

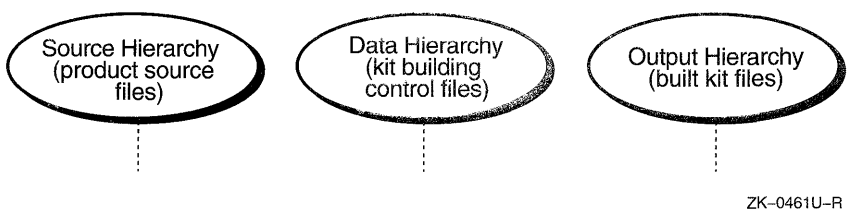
The following sections describe how to create kits that can be installed with the `setld` utility. The process of creating and packaging a `setld`-compatible kit is illustrated in Figure 8-4. In this figure, dashed boxes represent optional steps; for example, you do not have to create subset control programs if your kit requires no special handling when it is installed. The commands enclosed in ellipses are provided specifically for performing the indicated parts of the kit building process.

Figure 8-4: The Kit Building Process



Creating a kit requires the existence of three separate directory hierarchies, as shown in Figure 8-5. Each of these hierarchies is described in detail in the following sections.

Figure 8-5: Directory Hierarchies for a Kit



8.5.1 Creating a Source Hierarchy

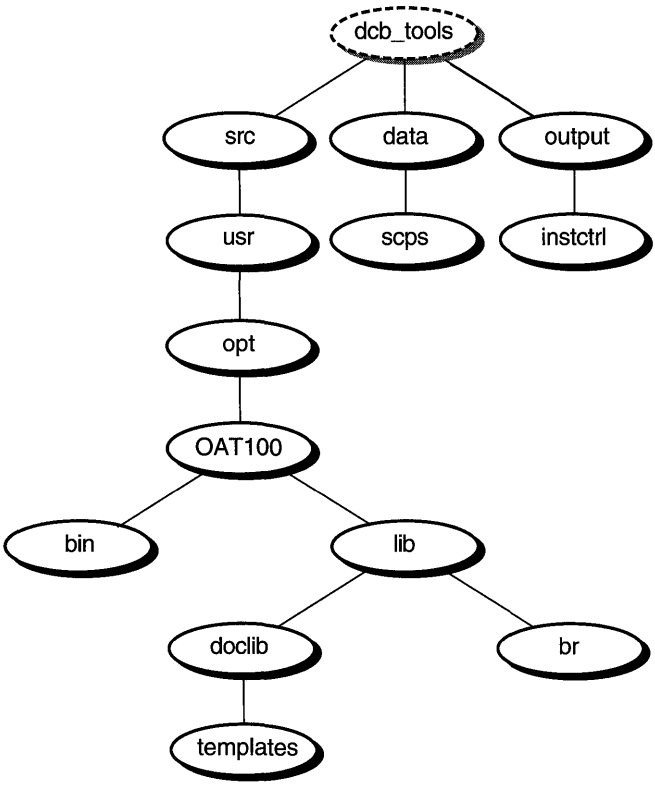
To build a `setld`-compatible kit, you must create a **source hierarchy**. A source hierarchy is a directory tree that exactly mirrors the hierarchy into which your finished kit will be installed by customers. This latter hierarchy is called the **target hierarchy**. You must place each file that is to become part of your kit into the appropriate directory in the source hierarchy. You can create the source hierarchy under any directory you choose. For example, the DCB kit contains files that are to be installed in the following directories:

- `/usr/opt/OAT100/bin`
- `/usr/opt/OAT100/lib/br`
- `/usr/opt/OAT100/lib/doclib/templates`

Figure 8-6 illustrates the complete hierarchy for the DCB kit. The `src` directory and the directories under it are the source hierarchy.

In this figure, the dashed directory, `dcb_tools`, represents the existing directory under which you would create the source hierarchy's directories as shown. The `src` directory you create represents the root directory in the target hierarchy on the customer's system; likewise, your `usr` directory represents `/usr` in the target hierarchy, and all the other directories in the source hierarchy are mapped to the customer's system in the same way. File names in the standard directory hierarchy, where users would normally expect the files to appear, will be linked symbolically to the actual files installed in the target hierarchy. For example, the command named `/usr/bin/attr` will exist as a link to `/usr/opt/OAT100/bin/attr`.

Figure 8-6: Directory Hierarchy for the DCB Kit



As shown in Figure 8-6, the name of the top-level product-specific directory, under the source hierarchy's `opt` directory, is made up of the product code and the version number. (See Section 8.5.2.2 for information about the product code and version number.) If the DCB kit included user-writable files, which should be placed under `/var`, there would also be a `/var/opt/OAT100...` hierarchy to contain those files. Digital recommends this convention for consistency among layered products.

After creating the source hierarchy, you must populate it with all the files that are to be part of the finished kit. You can choose any appropriate method for populating the source hierarchy; one common method is to create a makefile for use with the `make` command.

Note

File attributes, such as ownership and permissions, for all the files and directories in the source hierarchy must be exactly as they should be on the target hierarchy. Typically, this requirement means that you must be a superuser when populating the source hierarchy so that you can change these file attributes. Do not attempt to circumvent this requirement by setting file attributes in your subset control programs (SCPs); if a superuser on the target system runs the `fverify` command on your subsets, attributes that the SCPs have modified will be reset to the values they have in the kit's subset inventory files. (See Section 8.5.4.4.)

Under most circumstances, your kit should not include any files whose target pathnames exactly match those of existing system files. If you do, the kit's files will be installed in place of the existing files. In special cases, you can duplicate file names; see Section 8.5.3.

8.5.2 Creating the Kit Building Control Files

The `data` directory contains the following files that specify the contents of your kit and how it is to be organized:

- The master inventory file lists each of the files in the kit and defines which subset contains each file.
- The key file specifies the kit's attributes for the `kits` command. It describes the naming conventions and structure for the kit and provides high-level descriptions of the subsets.

The `data` directory also contains a subdirectory named `scps`; if your product includes subset control programs (SCPs), these programs are stored in the `scps` directory.

There is no specific requirement for the location of the `data` directory, but it is good practice to place it under the same directory in which you create the source hierarchy. Figure 8-6 illustrates the location of the DCB kit's `data` and `scps` directories.

8.5.2.1 Creating the Master Inventory

The master inventory for a kit specifies each file that is part of the kit. After you have populated the source hierarchy, you must create a master inventory in the `data` directory. The master inventory file's name consists of the product code and version, with the letters `mi` as a suffix; for example,

`OAT100.mi` is the master inventory file for the DCB kit. See Table 8-5 for a description of the correct form for the product code and version.

The master inventory file contains one record (line) for each file in the kit. Example 8-1 illustrates a portion of the master inventory for the DCB kit.

Example 8-1: Master Inventory File

```
0       .       RESERVED
0       ./usr/opt       RESERVED
0       ./usr/opt/OAT100        OATDCB100
0       ./usr/opt/OAT100/OATDCB.Links   OATDCB100
0       ./usr/opt/OAT100/OATDCBDOC.Links        OATDCBDOC100
0       ./usr/opt/OAT100/bin    OATDCB100
0       ./usr/opt/OAT100/bin/attr       OATDCB100
0       ./usr/opt/OAT100/bin/dcb.spr    OATDCB100
0       ./usr/opt/OAT100/bin/dcb_defaults       OATDCB100
0       ./usr/opt/OAT100/bin/dcb_diag.sed       OATDCB100
0       ./usr/opt/OAT100/bin/docbld     OATDCB100
0       ./usr/opt/OAT100/bin/unstamp    OATDCB100
0       ./usr/opt/OAT100/lib    OATDCB100
0       ./usr/opt/OAT100/lib/br OATDCB100
0       ./usr/opt/OAT100/lib/br/README.dcb      OATDCB100
0       ./usr/opt/OAT100/lib/br/attr.1  OATDCBDOC100
0       ./usr/opt/OAT100/lib/br/dcb.ps  OATDCBDOC100
0       ./usr/opt/OAT100/lib/br/docbld.1        OATDCBDOC100
0       ./usr/opt/OAT100/lib/br/unstamp.1       OATDCBDOC100
0       ./usr/opt/OAT100/lib/doclib     OATDCB100
0       ./usr/opt/OAT100/lib/doclib/templates   OATDCB100
0       ./usr/opt/OAT100/lib/doclib/templates/conv.braces       OATDCB100
   .
   .
   .
```

Each record in the master inventory consists of three fields, described in Table 8-4.

Table 8-4: Fields in Master Inventory Records

Field	Description
Flags	A 16-bit unsigned integer Bit 1 is the `v` (volatility) bit; when it is set, changes to the existing copy of the file can occur during kit installation. It is usually set for files such as `usr/spool/mqueue/syslog`. The remaining bits are reserved; possible values for this field are therefore 0 or 2.
Pathname	The dot-relative (`./`) pathname of the file

Table 8-4: (continued)

Field	Description
Subset identifier	The name of the subset containing the file You must not include standard system directories in your subsets. In the DCB master inventory example, several records specify directories that are part of the standard system hierarchy. Instead of a subset identifier, these records specify RESERVED; this keyword prevents `setld` from attempting to overwrite existing directories.

A subset is the smallest installable entity in a kit. It is up to you, the kit developer, to specify how many subsets your kit will have and what files each will contain; a good practice is to group files by related function or interdependence. For example, message libraries should be together in a subset to allow for easy localization and translation to other languages. Subset naming conventions are explained in Section 8.5.2.2. Example 8-1 illustrates a kit having two subsets. The OATDCB100 subset contains utilities and libraries and must be installed if the product is to be used. The OATDCBDOC100 subset contains the product's documentation and is not required to make the product function.

The first time you process a given kit, the master inventory file is empty. You can create this file with the `touch` command as follows:

```
% cd data
% touch OAT100.mi
```

For subsequent updates to the kit, the input file is the existing version of the master inventory.

Once you have a master inventory file, run the `newinv` utility. On the command line, specify the file name of the master inventory and the pathname of the source hierarchy's top-level directory. For example:

```
% newinv OAT100.mi ../src
```

The `newinv` utility performs the following tasks:

- Creates a backup file, *`inventory-file`*`.bkp`
- Finds all the file and directory names in the source hierarchy
- Produces the following three sorted groups of records:
  - Records containing pathnames only, representing files now present that were not in the previous inventory
  - Records representing files now present that were also present in the previous inventory

- Records that were in the previous inventory but for which files are not now present

- Leads you through a process to edit the third of these groups, deleting records for files which no longer belong in the kit
- Leads you through a process to edit the group of new records by adding the flag and subset fields (see Table 8-4)
- Merges the three groups of records and sorts the result to produce a finished master inventory file that matches the source hierarchy

The two editing steps in this procedure place you in either the `vi` editor or the editor specified by your `EDITOR` environment variable, so that you can make the required changes.

Caution

As indicated in Table 8-4, the files listed in the master inventory are given dot-relative pathnames. The `setld` utility normally works from the system's root directory, but the user can specify an alternative "root directory" with the `–D` option. For this reason, you should not use absolute pathnames in the master inventory.

Use extreme care when editing the master inventory; fields in this file must be separated by single tab characters, not by spaces.

8.5.2.2 Creating the Key File

The key file describes the high-level structure of the kit. This file resides in the `data` directory, and its name consists of the product code and version, with the letter `k` as a suffix. For example, `OAT100.k` is the key file for the DCB kit. Example 8-2 illustrates this key file. See Table 8-5 for a description of the correct form for the product code and version.

Example 8-2: Key File

```
#       Product-level attributes
#
NAME='Orpheus Authoring Tools'
CODE=OAT
VERS=100
MI=OAT100.mi
COMPRESS=1
#
#       Subset definitions
#
```

Example 8-2: (continued)

```
%%
OATDCB100        .        0        'Document Building Tools'
OATDCBDOC100     .        2        'Document Tools Documentation'
```

As shown in this example, the key file is divided into the following two sections separated by a line containing two percent signs (`%%`):

- The product attributes section describes the naming conventions for the kit and provides kit-level instructions for the `kits` command.
- The subset descriptor section describes each of the subsets in the kit and provides subset-level instructions for the `kits` command.

The product attributes section of the key file consists of several lines called **attribute-value pairs**. Table 8-5 describes each possible attribute-value pair. See Example 8-2 for examples of how these attributes are specified. Note that each attribute's name is separated from its value by an equal sign (=). Begin a comment line with a number sign (#).

Table 8-5: Key File Attributes Section

Attribute-Value Pair	Description
NAME	The product name; for example, 'Orpheus Authoring Tools' Enclose the product name in single quotation marks (') if it contains any spaces.
CODE	A unique product code consisting of three numbers or uppercase letters; the first character must be a letter. For example, OAT The following codes are reserved by Digital: DNP, DNU, EPI, FOR, LSP, ORT, OSF, SNA, UDT, UDW, UDX, ULC, ULT, ULX, UWS
VERS	A 3-digit version code; for example, 100 The `setld` utility interprets this version code as 1.0.0. The first digit should reflect the product's major release number, the second the minor release number, and the third the upgrade level, if any.
MI	The name of the product's master inventory file The master inventory file is created and maintained with the `newinv` utility.

Table 8-5: (continued)

Attribute-Value Pair	Description
ROOT	Not illustrated in the example, this optional attribute is reserved by Digital for the base operating system. For that use, it has a string value that names the root image file. Do not assign the `ROOT` attribute for a layered product.
COMPRESS	An optional flag that is set to 1 if you want to create compressed subset files Compressed files require less space on the distribution media (sometimes as little as 40% of the space required by uncompressed files), but they take longer to install than uncompressed files. If missing, this flag defaults to zero (`0`).

The subset descriptor section contains one line for each subset in the kit and cannot contain comment lines. Each line of this section consists of four fields separated with tab characters. Table 8-6 describes these fields; see Example 8-2 for an illustration of how they are coded. In this example, the OATDCB100 subset is mandatory; OATDCBDOC100, which contains the documentation, is optional.

Table 8-6: Key File Subset Descriptor Fields

Field	Description
Subset identifier	A character string up to 80 characters in length, composed of the product code (for example, OAT), a mnemonic identifying the subset (for example, DCB), and the 3-digit version code (for example, 100). All letters in the subset identifier must be uppercase.
Reserved	Must be a single period (`.`).
Flags	A 16-bit unsigned integer The lower 8 bits are used by Digital to convey information to `setld`. Bit 0 is the "sticky" bit, indicating when set that the subset cannot be removed. Bit 1 indicates when set that the subset is optional. Bits 2 to 7 are reserved. Bits 8 to 15 are undefined and can be used by your product's SCP.

Table 8-6: (continued)

Field	Description
Subset description	A short description of the subset, delimited by single quotation marks ('); for example, `'Document Building Tools'` **Note** The percent sign character (`%`) is reserved in this field and must not be used for layered products.

Section 8.5.3.3 explains how your SCP can use the unreserved bits in the Flags field.

8.5.3 Creating Subset Control Programs

Subset control programs (SCPs) perform special tasks beyond the basic installation managed by `setld`. If your kit requires no special installation handling, you do not need SCPs. The following are some of the reasons you might need an SCP:

- Some of your kit's files must be customized before the product will work properly.
- You want to offer the user the option of installing some of the files in a nonstandard location.
- Your kit depends on the presence of other products.
- You need to establish nonstandard permissions or ownership for certain files.
- Your kit requires changes in system files such as `/etc/passwd`.

All of these tasks can be performed by an SCP. Note that layered product kits designed according to the guidelines in Section 8.4.2 must have SCPs to create the required links.

As a general rule, SCPs are very short programs; if written as a shell script, an SCP should be under 100 lines in length. If your SCP is lengthy, it is likely that you are trying to make up for a deficiency in the architecture or configuration of the product itself.

The following sections describe how to write SCPs for your product.

8.5.3.1 Invoking a Subset Control Program

Your kit does not need to do anything to invoke its SCPs. The `setld` utility has built into it several points at which it automatically invokes the SCP for the subset currently being installed. At each of these points the SCP can take action if appropriate. If your kit has no SCPs, the `kits` command creates an empty SCP file for each subset.

When invoking an SCP, `setld` sets the `ACT` environment variable to a value that the SCP must use to determine what action it will take. In some cases, `setld` also calls the SCP with a command argument (`$1`) that modifies the `ACT` environment variable's meaning. Table 8-7 lists the possible settings of the `ACT` environment variable, the command arguments that can be used, and the conditions under which each combination of `ACT` and argument are used.

Table 8-7: Subset Control Program Invocation and Actions

ACT Setting	Command Argument	Description
`M`	`-l` `-x`	Before presenting the subset menu. At this time the SCP can decide whether it will permit its subset to be offered in the menu. The `-l` argument indicates that the operation is a subset load; `-x` is reserved by Digital. A return status of 0 (zero) allows the subset to be offered.
`PRE_L`		After presenting the menu, before loading the subset. At this time the SCP can take any action required to protect existing files. For example, the subset might contain files with names duplicating those of existing files. Subset dependency checking should be performed at this time. A return status of 0 (zero) allows the load operation to continue. **Note** Duplicating existing file names is usually considered poor practice. One example of a situation in which you might do so is for the installation of a kit that contains binary files that would normally be installed by other kits but which must be different if your kit is installed.
`POST_L`		After loading the subset. At this time the SCP can make any modifications required to subset files that will normally be protected from modification when the installation is complete, such as moving them to a different location. Link creation for layered products and subset dependency locking should be performed at this time.

Table 8-7: (continued)

ACT Setting	Command Argument	Description
C	INSTALL	After securing the subset. At this time the SCP can make "cleanup" modifications required for node-specific tailoring; for example, modifying `/etc/passwd`. Layered products' symbolic links cannot be created at this time.
V		For subset verification. At this time the SCP can perform tests to verify that the subset is installed correctly. (The `setld` utility verifies the size and checksum information for each file in the subset during loading.) For example, in a kit containing multiple subsets, the last subset's SCP could execute an Installation Verification Program (IVP) or suite of IVPs to ensure that the product works properly. The `setld` utility does not call the SCP for verification during the installation process; to invoke this portion of an SCP, use the –v option to `setld`.
C	DELETE	Before deleting the subset. At this time the SCP can make "cleanup" modifications to remove evidence of the subset's existence from the system; for example, removing a line that was added to the `passwd` file by the SCP during installation. Layered products' links cannot be removed at this time.
PRE_D		Before deleting a subset. At this time the SCP can reverse modifications made during the `POST_L` phase of installation, such as removing dependency locks and layered products' links, or restoring moved files to their default installation locations so that `setld` can delete them properly. A return status of 0 (zero) allows the deletion operation to continue.
POST_D		After deleting the subset. At this time the SCP can reverse modifications made during the `PRE_L` phase of installation.

An SCP can be written in any programming language, but you must take care that your SCPs can be executed on all platforms the kit can be installed on. If your product works on more than one hardware platform, you cannot write your SCP in a compiled language; for this reason, Digital recommends that you write your SCPs as scripts for `/sbin/sh`.

Depending on the tests it makes, your SCP could decide at some point to abort the installation or deletion of its subset. For example, if it checks for the existence of subsets upon which your product depends and fails to find one or more of them, the SCP can abort the process. To abort the installation or removal of the subset, the SCP must return a nonzero status to `setld` upon exiting from the particular phase for which it was called. If a status of zero (`0`) is returned, `setld` assumes that the SCP is satisfied that the

process should continue normally. See Example 8-3 for an illustration of how an SCP aborts installation.

8.5.3.2 Managing Subset Dependencies

Subset dependencies are conditions under which a given subset depends on the existence of one or more other subsets. This section explains dependencies and tells you how to manage them effectively.

Because `setld` is used for both installing and removing subsets, the system administrator could attempt to remove one or more subsets on which your product depends. Because those subsets do not in turn depend on your product's subsets, `setld` will normally remove them without question, leaving your product unable to work. You can prevent this inadvertent destruction of your product's environment by **locking** your subset's dependencies.

To make dependency management easier to implement, Digital provides a set of functions in the form of Bourne shell script code. These functions are contained in the file `/usr/share/lib/shell/libscp` and are listed in Table 8-8. The following sections describe how to use each of the user-callable functions.

Table 8-8: Dependency Management Routines

Routine	Description
`STL_ArchAssert`	Verify that the hardware architecture of the machine matches the product's target architecture. This function is not actually a dependency function; it is included in `libscp` for convenience.
`STL_DepInit`	Initialize dependency checking conditions. This function must be executed before any tests are made.
`STL_DepEval`	Evaluate a dependency expression, returning 0 (zero) if the expression is satisfied and 1 if not.
`STL_LockInit`	Initialize dependency locking conditions. This function must be executed before any locking or unlocking is done.
`STL_DepLock`	Add the new subset to the lock lists for all the named subsets.
`STL_DepUnLock`	Remove the new subset from the lock lists for all the named subsets.

You should not include a copy of these functions explicitly in your SCP because such a design prevents your kit's benefiting from enhancements or

bug fixes made in future releases. Use the shell's dot (source) command to call the functions. The SCP illustrated in Example 8-3 uses this technique.

8.5.3.2.1 Dependency Expressions – The dependency management functions use **dependency expression** to examine conditions on the system. A dependency expression is a postfix logical expression in Backus-Naur form (BNF) that describes conditions on which the new subset depends. Dependency expressions have the following syntax:

depexp ::= *wc_subset_id*
| *depexp* **not**
| *depexp depexp* **and**
| *depexp depexp* **or**

Dependency expressions are recursive left to right; they are processed using conventional postfix techniques. The symbol *wc_subset_id* represents a subset identifier that can contain file name expansion characters (asterisks, question marks, or bracketed sets of characters) as in the following example:

```
OAT[RV]DOA*2??
```

The following operators are used:

- `and`

  Requires two dependency expressions. Satisfied if both expressions are satisfied.

- `or`

  Requires two dependency expressions. Satisfied if at least one of the expressions is satisfied.

- `not`

  Requires one dependency expression. Satisfied if the expression is not satisfied.

The following are valid dependency expressions:

```
SUBSETX??0
SUBSETY200 not
SUBSET[WX]100 SUBSETY200 and
SUBSETX100 SUBSETY200 or
SUBSETX100 SUBSETY200 and SUBSETZ300 or not
```

The last of these expressions evaluates as follows:

1. The `and` operator is satisfied if both SUBSETX100 and SUBSETY200 are present.
2. The `or` operator is satisfied if the `and` operator was satisfied or if SUBSETZ300 is present.

3. The `not` operator is satisfied only if the `or` operator was not satisfied.

Hence, this expression is satisfied only if the combination of SUBSETX100 and SUBSETY200 is not present and SUBSETZ300 is not present.

8.5.3.2.2 Using the STL_DepInit Routine – Use `STL_DepInit` in the `PRE_L` phase (see Table 8-7) to establish objects used by the `STL_DepEval` function. Before you use `STL_DepEval` to check your subset's dependencies, you must execute `STL_DepInit` once. This function has no arguments and returns no status.

8.5.3.2.3 Using the STL_DepEval Routine – Use `STL_DepEval` in the `PRE_L` phase (see Table 8-7) after calling `STL_DepInit`. This function requires a dependency expression as an argument. You can use as many invocations of `STL_DepEval` as you need to verify that all your subset dependencies are met.

8.5.3.2.4 Using the STL_ArchAssert Routine – Use `STL_ArchAssert` during the `M` phase (see Table 8-7); if the function is not satisfied, the subset will not be presented for installation. This function requires a single argument, a keyword naming the target architecture. This keyword is the value returned by the `machine`(1) command; for Digital's RISC processors, the keyword is `mips`.

8.5.3.2.5 Using the STL_LockInit Routine – Use `STL_LockInit` in the `POST_L` and `PRE_D` phases (see Table 8-7) to establish objects used by the `STL_DepLock` and `STL_DepUnLock` functions. Before you use `STL_DepLock` or `STL_DepUnLock` to manipulate subset locks, you must execute `STL_LockInit` once. Note that because locking and unlocking are managed by different invocations of your SCP, `STL_LockInit` must appear in both the `POST_L` and `PRE_D` phases. You should code two instances of `STL_LockInit` rather than calling it once before you make a decision based on the value of the `ACT` environment variable. This function has no arguments and returns no status.

8.5.3.2.6 Using the STL_DepLock Routine – Use `STL_DepLock` in the `POST_L` phase (see Table 8-7) to add the new subset's name to the lock lists for each of the subsets named as arguments. (You can use dependency expressions as arguments.) The name of the new subset is the first argument to `STL_DepLock`; for example, the following line places `OATDCB100` in the `OATTOOLS100.lk` and `OATBASE2??.lk` files:

```
STL_DepLock OATDCB100 OATTOOLS100 OATBASE2??
```

Lock lists are contained in the *subset-id*.lk files in the /usr/.smdb. directory.

8.5.3.2.7 Using the STL_DepUnLock Routine – Use STL_DepUnLock in the POST_D phase (see Table 8-7) to remove the new subset's name from the lock lists for each of the subsets named as arguments.

8.5.3.3 Using Control File Flag Bits

As indicated in Table 8-6, you can use bits 8 to 15 in the key file's Flags attribute to specify special subset-related information. The SCP can read these bits from the subset control file, into which this information is placed when the kit is built. During installation, the setld utility moves the control file to the ./usr/.smdb. directory; the SCP can read the file as needed. The SCP should look for a line like the following example from the OATDCBDOC100.ctrl file:

```
FLAGS=34816
```

The value of this attribute is expressed as a decimal integer. You can use the BitTest shell function, contained in the file /usr/share/lib/shell/BitTest, to test an individual bit. The following sample /sbin/sh code tests bit 11 of the FLAGS attribute for the OATDCBDOC100 subset:

```
#! /sbin/sh

. /usr/share/lib/shell/BitTest

flags='sed -n '/FLAGS=/s///p' usr/.smdb./OATDCBDOC100.ctrl'
BitTest $flags 11 && {
   •
   •
   •
}
```

8.5.3.4 An Example Subset Control Program

Example 8-3 shows an SCP for the DCB product that is illustrated throughout this chapter. This SCP illustrates one correct method for obtaining the value of the ACT environment variable and acting on that value. It also shows how to create and remove the links for a layered product in one of the opt directories by calling an independent script that is unique to the subset. (The linking script is shown in Example 8-4.)

Note that the example SCP provides no code for several of the possible conditions under which it can be invoked. The case statement that chooses an action simply exits with zero status in these undetected cases, and setld

continues normally. Do not include a wildcard in your SCP's option parsing routine; write code only for the cases the SCP actually handles.

Caution

The example SCP uses the shell's source command to include the dependency and subset locking functions described in Section 8.5.3.2. Although it is mechanically possible to include a copy of these functions explicitly, you should not do so because such a design prevents your kit's benefiting from enhancements or bug fixes made in future releases.

Example 8-3: Sample Subset Control Program

```
#! /sbin/sh
#
# Subset Control Program for OATDCB??? subset

# INCLUDE DEPENDENCY ROUTINES

. /usr/share/lib/shell/libscp     1

# BEGIN EXECUTION HERE

SubSet=OATDCB100
Desc="Document Building Tools"
kroot=usr/opt/OAT100
case $ACT in      2
  PRE_L)      3
    STL_DepInit
    STL_DepEval OATTOOLS??? || {
      oops="$oops
   Orpheus Authoring Tools (OATTOOLS)"
    }
    STL_DepEval OATBASE2?? || {
      oops="$oops
   Orpheus Authoring Base Tools, Version 2.0 or later (OATBASE)"
    }
    [ "$oops" ] && {
      echo "
The $Desc require the existence of the
following uninstalled subset(s):
$oops

Please install these subsets before retrying the DCB installation.
" >&2
      exit 1
    }
    ;;
  POST_L)    4
    $kroot/OATDCB.Links INSTALL
    STL_LockInit
    STL_DepLock $SubSet OATTOOLS??? OATBASES2?? and
    ;;
  C)
    case $1 in      5
```

Example 8-3: (continued)

```
      INSTALL)                                6
        echo "
Installation of the $Desc ($SubSet)
subset is complete.

Before using the tools in this subset, please read the README.dcb
file located in the /usr/lib/br directory, for information on the
kit's contents and for release information.
"
        ;;
    esac
    ;;
  PRE_D)        7
    $kroot/OATDCB.Links DELETE
  POST_D)       8
    STL_LockInit
    STL_DepUnLock $SubSet OATTOOLS??? OATBASE2?? and
    ;;
esac
exit 0     9

#       End of Subset Control Program script.
```

The following list describes the activities performed by the annotated code segments in this example:

1 This command reads in the dependency checking and locking routines. Do not include a copy of these routines explicitly in your SCP.

2 This `case` statement processes the `ACT` environment variable to select the action the SCP will take when called by `setld`.

There is no case for the `M` phase. The OATDCB100 subset's SCP takes no action when called to determine if it will present itself as a candidate for installation in the `setld` subset menu. Actions that could be taken by your SCP might include selecting between two functionally identical subsets whose only difference is that they are intended for different hardware platforms. Similarly, the `case` statement does not recognize any other possible case for which the SCP has no code.

3 The `PRE_L` code tests to ensure that subsets on which the OATDCB100 subset depends are installed. If they are not, the SCP describes the missing subsets and returns nonzero status to `setld`, which will abort the installation of this particular subset. If multiple subsets are being installed, each is treated individually.

4 The `POST_L` code creates symbolic links for the subset by running an external script that is associated with this subset. This script reads its command line argument to determine whether to create or remove the links. An example of such a script is shown in Example 8-4.

After creating the links, the SCP secures the subset by locking the subsets on which it depends in order to ensure that they are not deleted without warning the user of potential problems.

5 This `case` statement handles the subset installation and deletion configuration code; it is invoked when the `ACT` environment variable is set to `C`.

6 For this subset, there is no special file management or other cleanup to be performed. The `INSTALL` code notifies the person running `setld` that the subset's installation is complete. It also refers the installer to a text file containing important information about the kit and its contents. This SCP takes no action for the `C DELETE` case.

7 The `PRE_D` code calls the external script to remove the links that were created during the `POST_L` phase.

8 The `POST_D` code unlocks the subsets on which OATDCB100 depends.

9 This statement ensures that the SCP returns success status to `setld` for each successful action and for all of the possible cases that the SCP does not handle. Do not code `exit 0` statements elsewhere in your SCP.

8.5.3.5 Creating Symbolic Links for Layered Products

As indicated in Section 8.4.2, layered products' files should be installed in the `/usr/opt` and `/var/opt` areas and accessed by means of symbolic links in the traditional UNIX directory hierarchy. These symbolic links, referred to as "forward" links, must be created during the `POST_L` phase, after the referent files are in place. Do not try to create these links during the `C INSTALL` phase because the `/usr` file system is not guaranteed to be writeable at that time. If your product includes links in `/var`, create these links also in `POST_L`. To maintain symmetry, you must remove links during `PRE_D`, not during `C DELETE`.

When you import a kit using NFS, you do not create the forward links on the client by running the SCP. Instead, these links must be made by running an independent program that can also be called (on the server) by the SCP, as illustrated in Example 8-3. Section 8.5.3.5.1 describes how to create forward links.

As described in the previous paragraph, symbolic links for layered products are typically created in the traditional UNIX directories to refer to files that are actually in the layered product areas (`/usr/opt` and `/var/opt`). These links are relatively straightforward, as shown in Example 8-4.

Under certain circumstances, however, you might need to create links within your product's directories in the layered product areas that refer to files in the traditional hierarchy. Such "backward" links must be created with especial care because the layered product directories can themselves be symbolic links. This means that you cannot rely on knowing in advance the correct number of dot-dot levels (`../`) to include in the `ln` commands for your backward links.

For example, /var is frequently a link to /usr/var.

When a kit is installed on an NFS server, all the backward links are made in the server's kit area. Then, when that area is exported to clients, the links are already in place for the client; clients do not need to create any backward links.

Note

NFS clients importing products with backward links must have directory hierarchies that exactly match those on the server; otherwise, the backward links will fail.

See Section 8.5.3.5.2 for information on creating backward links.

8.5.3.5.1 Creating Standard (Forward) Symbolic Links – To provide for NFS exporting of your kit, your SCP should manage its forward links by calling a separate script that can also be executed from the command line. This design allows the administrator of an NFS client to import the product and then create the links by executing the linking script.

The linking script should be located in the top-level product-specific directory so that it will be available to be run on the client. (For the DCB kit, this directory is /usr/opt/OAT100; see the master inventory in Example 8-1.) The example SCP shown in Example 8-3 works in this way, calling the linking script illustrated in Example 8-4.

Example 8-4: Sample Link Control Program

```
#! /sbin/sh
#
# Link Control Program for OATDCB??? subset

kroot=usr/opt/OAT100

case $1 in      [1]
  INSTALL)                          [2]
    for i in 'ls $kroot/bin'; do
      ln -s ../../$kroot/bin/$i usr/bin/$i 2>&-    [3]
    done
    for i in 'ls $kroot/lib/br'; do
      ln -s ../../../$kroot/lib/br/$i usr/lib/br/$i
    done
    [ -d usr/lib/doclib/templates ] || mkdir -p usr/lib/doclib
    for i in 'ls $kroot/lib/doclib/templates'; do
      ln -s ../../../../$kroot/lib/doclib/templates/$i \
      usr/lib/doclib/templates/$i
    done
    ;;
  DELETE)                           [4]
    for i in 'ls $kroot/bin'; do
      ls -l usr/bin/$i | grep -s -e '->' && rm -f usr/bin/$i    [5]
    done
```

Example 8-4: (continued)

```
    for i in 'ls $kroot/lib/br'; do
      rm -f usr/lib/br/$i
    done
    for i in 'ls $kroot/lib/doclib/templates'; do
      rm -f usr/lib/doclib/templates/$i
    done
    rmdir usr/lib/doclib/templates 2>&-
    rmdir usr/lib/doclib 2>&-
    ;;
esac
exit 0

#       End of Link Control Program script.
```

The following list describes the activities performed by the annotated code segments in this example:

1 This `case` statement processes the command-line argument in the same way as the SCP handles its own command-line argument during the `C INSTALL` and `C DELETE` phases. In this example link control program, the argument is also given a value of `INSTALL` or `DELETE`. Do not confuse this argument with the SCP's argument; the link control program is called at `POST_L` and `PRE_D`.

2 If the argument is `INSTALL`, the link control program creates symbolic links in the standard system directories in which the files are to appear (`usr/bin` and `usr/lib/br`). Note the use of two dots (`..`) to indicate a parent directory. Because the `usr/bin` directory, for example, is two levels below the root directory, two levels of parent directories must be named in the symbolic links that are placed there.

   The DCB kit checks for the existence of, and if necessary creates, the `usr/lib/doclib` and `usr/lib/doclib/templates` directories, which are nominally new. It then creates links for the files in this new area.

3 The DCB kit includes a file that duplicates the function of a file that is part of another layered product kit, which might not be present. If the other kit is present, the link command will produce an error when it tries to create the link for the duplicated file. Redirecting error output prevents the user from seeing this meaningless error message.

4 The `DELETE` code removes the links and directories that were created by the `INSTALL` code; the files found in the kit's actual location provide the names of all the links that should exist.

5 This line verifies that the program files it is removing are actually links to the DCB kit's files. This design prevents the SCP from removing the duplicated file described in a previous step if the copy in `/usr/bin` belongs to the other kit rather than to the DCB kit.

8.5.3.5.2 Creating Backward Links – Backward links should be created by the SCP, not by the link management program, so that installation on an NFS client will not attempt to overwrite the existing backward links in the server's kit areas. (You do not run the SCP on an NFS client.) As with forward links, your SCP should create and remove backward links in the `POST_L` and `PRE_D` phases, respectively.

To simplify the creation of backward links, the `libscp` library provides two functions in addition to those described in Section 8.5.3.2; these additional functions are listed in Table 8-9 and described in the sections following the table.

Table 8-9: Routines Assisting with Backward Link Creation

Routine	Description
`STL_LinkInit`	Initialize an internal variable for use when creating backward links. This function must be executed before using `STL_LinkBack`.
`STL_LinkBack`	Creates one backward link. As arguments, requires the name of the file and the two directories involved.

8.5.3.5.3 Using the STL_LinkInit Routine – Use `STL_LinkInit` in the `POST_L` phase (see Table 8-7) to establish an internal variable used by the `STL_LinkBack` function. Before you use `STL_LinkBack` to create a link, you must execute `STL_LinkInit` once. This function has no arguments and returns no status.

8.5.3.5.4 Using the STL_LinkBack Routine – Use `STL_LinkBack` in the `POST_L` phase (see Table 8-7) to create a valid symbolic link from your product area (`/usr/opt`*`product-id`* or `/var/opt`*`product-id`*) into the traditional UNIX hierarchy. You can use `STL_LinkBack` repeatedly to create as many links as required; before you use `STL_LinkBack`, you must execute `STL_LinkInit` once. This function returns no status. It has the following three arguments, which must be in this order:

1. The name of the file to link
2. The dot-relative path of the directory where the file actually resides
3. The dot-relative path of the directory in which the link is to be placed

No special action is required to remove a link created by `STL_LinkBack`; you can simply use the `rm` command. Example 8-5 uses `STL_LinkInit` and `STL_LinkBack` in the `POST_L` phase to create a link named

/usr/opt/OAT100/lib/dcb_users that refers to the real file /etc/dcb_users, and removes the link in the PRE_D phase.

Example 8-5: Example of Backward link Creation

```
#! /sbin/sh
   •
   •
   •
case $ACT in
   •
   •
   •
  POST_L)
   •
   •
   •
    STL_LinkInit
    STL_LinkBack dcb_users ./etc ./usr/opt/OAT100/lib
   •
   •
   •
    ;;
  PRE_D)
   •
   •
   •
    rm -f ./usr/opt/OAT100/lib/dcb_users
   •
   •
   •
    ;;
   •
   •
   •
esac
```

8.5.4 Building Your Kit

The kit building procedure, performed by the `kits` command, creates subsets and control files that are placed into a directory that you specify. As with the `data` directory, there is no specific requirement for the location of the output directory, but it is good practice to place it under the same directory in which you create the source hierarchy. Create your output directory with the `mkdir` command. Under the output directory, create a second directory named `instctrl` to hold installation control files created by the `kits` command. (If you do not create the `instctrl` directory, the `kits` command will create it for you.) Figure 8-6 illustrates the DCB kit's output directory hierarchy.

After you create the output directories, change to the `data` directory and create the kit files by running the `kits` command. This command requires three arguments: the key file name, the pathname for the source hierarchy,

and the pathname for the output directory. For example, the DCB kit is built with the following command:

```
% kits OAT100.k ../src ../output
```

The `kits` command reports its progress as it creates the subsets. If you have specified the COMPRESS attribute in the key file, `kits` compresses each subset. After creating the subsets, `kits` creates the installation control files (of which *subset-id*`.ctrl` is one), placing them in the `instctrl` directory. Then `kits` creates a file named `INSTCTRL`, containing a `tar` image of all the control files. This file is placed in the output directory. The subset files and the `INSTCTRL` file are constituents of the final kit.

The control files created by `kits` are described in Table 8-10, and the following sections describe the contents of the files.

Table 8-10: Control Files in the instctrl Directory

File	Description
product-id`.comp`	Compression flag file This empty file is created only if you specified the COMPRESS attribute in the key file. Its presence signals to `setld` that the subset files are compressed. The DCB kit's compression flag file is named `OAT100.comp`.
product-code`.image`	Image data file This file contains size and checksum information for the subsets.
subset-id`.ctrl`	Subset control file This file contains `setld` control information. There is a control file for each subset.
subset-id`.inv`	Subset inventory file This file contains an inventory of the files in the subset. Each record describes one file. There is an inventory file for each subset.
subset-id`.scp`	Subset control program (SCP) If you have created SCPs for your kit, these files are copied from the `scps` directory to `instctrl`. There is one SCP for each subset; if you have not created an SCP for a given subset, `kits` creates a blank file.

8.5.4.1 The Compression Flag File

The `setld` utility uses the presence of the compression flag file (*`product-id`*`.comp`) to determine whether the subset files are compressed. The compression flag is an empty file whose name consists of the product code and the version number, with the string `comp` as a suffix; for example, `OAT100.comp`.

8.5.4.2 The Image Data File

The `setld` utility uses the image data file to verify that the subset images it loads from the installation media are uncorrupted before the actual installation process begins. The image data file's name consists of the product's unique 3-letter name with the string `image` for a suffix. The image data file contains one record for each subset in the kit. The following example illustrates `OAT.image`, the image data file for the DCB kit:

```
15923     70 OATDCB100
24305    400 OATDCBDOC100
```

Table 8-11 describes the three fields in each record.

Table 8-11: Image Data File Fields

Field	Description
Checksum	The modulo-65536 (16-bit) checksum of the subset file (after compression, if the file is compressed)
Size	The size of the subset file in kilobytes (after compression, if the file is compressed)
Subset identifier	The product code, subset mnemonic, and version number

See Table 8-5, in Section 8.5.2.2, for a description of the correct form of the product code, subset mnemonic, and version number.

8.5.4.3 The Subset Control Files

The `setld` utility uses the subset control files as a source of descriptive information about subsets. A control file for each subset contains the following information:

- Descriptive product name and subset identifier
- Descriptive subset identifier
- Disk volume identification information, consisting of two colon-separated integers (the volume number of the disk containing the subset archive and the number of diskettes required to contain the subset archive)

- Tape volume number and the subset's location on the tape, consisting of two colon-separated integers (the volume number of the tape containing the subset archive and the file offset at which the subset archive begins)

  On tape volumes, the first three files are reserved for a bootable operating system image and are not used by `setld`. An offset of 0 (zero) indicates the fourth file on the tape. The fourth file is a `tar` archive named `INSTCTRL`, containing the kit's installation control files (listed in Table 8-10).
- Dependency list (reserved – see Table 8-6)
- Subset control flag bits

The following example illustrates `OATDCBDOC100.ctrl`, the control file for the DCB kit's OATDCBDOC100 subset:

```
NAME='Orpheus Authoring Tools OATDCBDOC100'
DESC='Document Tools Documentation'
NVOLS=1:2
MTLOC=1:1
DEPS="."
FLAGS=34816
```

8.5.4.4 The Subset Inventory Files

Each subset's inventory file describes each file in the subset, listing its size, checksum, permissions, and other information. This information is generated by the `kits` command and reflects the exact state of the files as they were in the source hierarchy from which the kit was built. It is used by `setld` to duplicate that state, thus transferring an exact copy of the source hierarchy to the customer's system. Example 8-6 shows the inventory file, `OATDCBDOC100.inv`, for the DCB kit's OATDCBDOC100 subset. The backslashes (\) in this example indicate line continuation and are not present in the actual file.

Example 8-6: Sample Subset Inventory File

```
0       983     01851   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/attr.1  none    OATDCBDOC100
0       424997  63356   1065    10      100644  4/15/91 100     f\
       ./usr/opt/OAT100/lib/br/dcb.ps  none    OATDCBDOC100
0       7283    03448   1065    10      100644  4/15/91 100     f\
       ./usr/opt/OAT100/lib/br/docbld.1        none    OATDCBDOC100
0       6911    37501   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/docbld.5        none    OATDCBDOC100
0       985     41926   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/unstamp.1       none    OATDCBDOC100
```

Each record of the inventory is composed of 12 fields separated by tab characters. Table 8-12 describes the contents of these fields.

Table 8-12: Subset Inventory Field Descriptions

Field	Name	Description
1	Flags	A 16-bit unsigned integer Bit 1 is the `v` (volatility) bit; when it is set, changes to the existing copy of the file can occur during kit installation. It is usually set for files such as `usr/spool/mqueue/syslog`. The remaining bits are reserved; possible values for this field are therefore 0 or 2.
2	Size	The actual number of bytes in the file
3	Checksum	The modulo-65536 (16-bit) checksum of the file
4	uid	The user ID of the file's owner
5	gid	The group ID of the file's owner
6	Mode	The 6-digit octal representation of the file's mode
7	Date	The file's last modification date
8	Revision	The version code of the product that includes the file
9	Type	A letter that describes the file: `b` – Block device `c` – Character device `d` – Directory containing one or more files `f` – Regular file. For regular files with a link count greater than one, see file type `l`. `l` – Hard link. There are other files in the same inventory which have the same inum. The first of these files in ASCII collating sequence is listed in the referent field. `p` – Named pipe (FIFO) `s` – Symbolic link
10	Pathname	The dot-relative (`./`) pathname of the file
11	Referent	For file types `l` and `s`, the path to which the file is linked; for types `b` and `c`, an integer representing the major and minor numbers of the device; for all other types, `none`
12	Subset identifier	The name of the subset containing the file

8.5.5 Transferring Your Kit to Distribution Media

Before you can transfer your kit to distribution media, you must have or create the following files:

- A file named SPACE in the output directory

  This file is a 10240-byte file whose contents are reserved by Digital. At present, it is empty. This file is a placeholder for tape records. To create a SPACE file, change to the output directory and enter the following command:

  ```
  % tar cf SPACE /dev/null
  ```

- An /etc/kitcap file

  This file contains one record for each kit your system can produce. See the kitcap(4) reference page for a description of the format and contents of a kitcap record. The order in which you list the subsets in the /etc/kitcap record defines their order on the distribution media.

8.5.5.1 Building a Kit on Magnetic Tape

Use the gentapes command to create a kit on magnetic tape. The syntax of the gentapes command is as follows:

gentapes [**-w** | **-v**] [*node*:] *product-id tape-device*

Note that there must be no space between the *node* and *product-id* names. For example:

```
% gentapes vizier:OAT100 /dev/nrmt0h
```

The –w option specifies that gentapes write the tape without verifying it; the –v option specifies that the command verify a tape without writing it first. If neither option is specified, gentapes writes the tape, rewinds it, and verifies its contents.

If you specify a node, gentapes looks for the output directory on that node. It also expects that the kit will be specified in that node's /etc/kitcap file and will be located on that node. If you do not specify a remote node, gentapes looks on your own system. You can use NFS file sharing to mount the kit files remotely on a system with the required tape drive.

8.5.5.2 Building a Kit on a Disk

Use the gendisk command to create a kit on a disk. The syntax of the gendisk command is as follows:

gendisk [*node*:] *product-id target-disk*

Note that there must be no space between the *node* and *product-id* names. For example:

```
% gendisk vizier:OAT100 /dev/rz0c
```

When you run `gendisk`, the command creates a file system on the target disk partition specified in the `/etc/kitcap` entry and then transfers the kit to that partition.

The `gendisk` command supports the optional `node` specification as described in Section 8.5.5.1 for `gentapes`.

Typically, you must add a SCSI disk device with removable media to your system in order to create disk kits. To add a new device type, edit the `/etc/disktab` file to supply the required disk specification. (See the `disktab`(4) reference page for information on the disk specification format.) Then run the `MAKEDEV` command to create the device special files. If adding the new disk drive exceeds the number of drives your system is configured for, you must edit the configuration file and rebuild the kernel. See the *System Administration.*

8.5.6 Installing and Distributing Kits on a Network

Inherent in the structure of a `setld` kit is the ability to copy and install kits locally or across a network without having to place them on distribution media, thereby saving the cost of multiple copies of the kit. If you have been granted the right to copy kits you have purchased, you can reproduce those kits and offer them as well. You can install kits in the following ways:

- Locally, from the output hierarchy
- Remotely, by explicit copying from the output hierarchy
- Remotely, by creating a network installation script
- Remotely, by exporting the output hierarchy

The kits that the `gendisk` command builds on distribution media are exact duplicates of the output hierarchy created by the `kits` command. The `setld` command's arguments consist of a *location* and, optionally, one or more *subset-id* names. The *location* argument is not limited to mount points such as `/dev/rrz3c` or `/mnt`; it can specify any desired directory. For example, after the DCB kit is built, a superuser on your

system can change to the `dcb_tools` directory and install the kit with the following command:

```
# setld -l output
```

To install a kit remotely, you can create a directory hierarchy on the target system and use network copy commands to populate it. Then you can use the `setld` command on the remote system to install the kit locally there.

To simplify the task of the remote system administrator, you can create an intelligent script that remote system administrators can copy and execute on their own systems. Such a script should do at least the following things:

1. Identify itself and describe the kit that it will install.
2. Find a file system with enough space for the kit.
3. Create a kit directory hierarchy in the file system found by the previous step, and populate it by copying the kit files from the system where they are stored.
4. Verify that the copied files appear to be uncorrupted.
5. Run `setld` to install the kit.

If you prefer not to leave a copy of the kit on the remote system, you can install the kit remotely by using NFS file sharing to export the directory containing the kit to a target system. Mount the kit directory on the target system and use `setld` normally.

Glossary

This glossary defines terms used in this manual. Each definition is keyed to the topic to which it applies.

attribute-value pair
: In a software kit's key file, a line specifying the name and value for a single attribute of the kit. Controls how the kit is built by the `kits` command and how it is installed by the `setld` utility.

check in
: In the Revision Control System (RCS), to store a file or revision in the RCS library.

check out
: In the Revision Control System (RCS), to retrieve a file or revision from the RCS library.

collating symbol
: In a regular expression (RE), a name that defines a particular subset of the available characters, such as lowercase characters, in a collating sequence that uses multicharacter strings to represent single characters.

delta
: In an RCS or SCCS file, the set of changes that constitute a specific version of the file.

dependency expression
: In a subset's subset control program (SCP), a Backus-Naur form (postfix) logical expression consisting of subset identifiers and relational operators to describe the current subset's relationship to the named subsets. See **dependency, subset**.

dependency file
: See **dependent**.

dependency line
: In the `make` utility, a line in the description file that describes the dependents on which a given target depends.

dependency, subset
: The condition in which a given subset requires the presence, or lack thereof, of other subsets in order to function properly. Evaluated by a subset's SCP under control of the `setld` utility.

dependent
: Also called a **dependency file**. In the `make` utility, an entity on which a file to be built (the **target**) depends. A source file is a dependent of an object module.

field
: In `awk`, one element of an input record; fields are separated by a field separator, which can be specified and is by default any amount of white space. The beginning and end of the record are also field separators. See **record**.

field variable
: In `awk`, a variable that is a field of the input record; field variables can be manipulated as any other variable.

g-file
: In the Source Code Control System (SCCS), the file whose contents are used to create the s-file or to apply a delta to it.

ID keyword
: In the Source Code Control System (SCCS), a symbol composed of a single letter enclosed by percent signs (`%`). In the Revision Control System (RCS), a symbol composed of a keyword name enclosed by dollar signs (`$`). In expanded form, a keyword provides identification information about the file, such as its date, version number, or name.

layered product
: In `setld` and product kit development, an optional software product designed to be installed as an added feature of the DEC OSF/1 system.

lexical analyzer
: A program or program fragment for analyzing input and assigning elements of it to categories to assist in parsing the input. See **parser**. The `lex` program assists in the creation of lexical analyzers.

locking
: In software installation by the `setld` utility, the act of inserting a new subset's name in the lock file of an existing subset so that an attempt to remove the latter subset will flag the user with a dependency warning. In a version control system, the creation and use of information flagging a version control file as being checked out for editing.

locking mechanism
: In a version control system, a way to prevent overlapping and concurrent changes to a file. SCCS uses p-files to indicate which files are currently out for editing; RCS creates locks by editing the RCS-file to insert lock information.

macro definition
: For the `m4` macro processor or the `make` utility, a statement creating a macro name and defining the text and argument substitutions for which the macro stands.

operator
: In regular expressions (REs), a character that is interpreted to mean something other than its literal meaning. For example, a pair of brackets (`[ ]`) form an operator that enables a single-character match on any one of the characters enclosed by the brackets.

p-file
: In the Source Code Control System (SCCS), a lock file whose presence indicates that the s-file of the same name is currently being edited.

parser
: A program or program fragment for interpreting input and determining how to act upon it. The `yacc` program assists in the creation of parsers.

pattern space
: In the `sed` editor, the range of lines currently being edited; the pattern space is selected by an address or pair of addresses.

RCS-file
: In the Revision Control System (RCS), a file stored in the RCS library, containing the text of the original file and the list of deltas that have been applied to it.

RCS library
: In the Revision Control System (RCS), the directory in which RCS-files are stored.

record
: In `awk`, The information between two consecutive occurrences of the record separator, which can be specified and is by default a newline character. For most purposes, a record can be thought of as a line from the input file. The beginning and end of the file are also record separators.

s-file
: In the Source Code Control System (SCCS), a file stored in the SCCS library, containing the text of the original file and the list of deltas that have been applied to it.

SCCS library
: In the Source Code Control System (SCCS), the directory in which SCCS s-files and p-files are stored.

script
: In the `sed` editor, a list of editing commands to be applied to the input file.

SID
: In the Source Code Control System (SCCS), the numeric identification applied to a particular delta.

source hierarchy
: For building software kits, the directory tree and files that are to be compiled by the `kits` command into subsets for a kit.

subset
: The smallest installable component of a software kit for the `setld` utility. Contains files of any type, usually interrelated in some way.

target
: Also called a **target file**. In the `make` utility, an entity to be built from its **dependents**. An executable program is a target that is built from one or more object modules.

target hierarchy
: For building software kits, the directory tree into which a software kit is placed by the `kits` command.

token
: For the `m4` macro processor, a recognizable entity that can be a macro name. A token consists of alphanumeric characters delimited by nonalphanumeric characters and cannot contain other tokens.

  For `lex`-generated lexical analyzers and `yacc`-generated parsers, the smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax.

version control file
: In a version control system, a file that consists of original text and a set of revisions (deltas) that have been made to it. In RCS, this file is called an RCS-file; in SCCS, an s-file.

version control library
: A directory that contains files that are organized and maintained under a version control system such as RCS or SCCS.

version control system
: A software tool that aids in the organization and maintenance of file revisions and configurations. In particular, it automates the storing, logging, retrieval, and identification of revisions to source programs, documentation, and data files.

younger file
: For the `make` utility, a dependency file that has changed more recently than its target.

Index

Special Characters

$
See dollar sign
&
See ampersand
()
See parentheses
*
See asterisk
+
See plus sign
.
See period
/
See slash
:
See colon, yacc
;
See semicolon
<>
See angle brackets
?
See question mark
@
See at sign
[ ]
See brackets
\
See backslash
^
See circumflex
{ }
See braces
|
See vertical bar

A

ACT environment variable, 8–28t, 8–28
action
awk, 2–11, 2–3
lexical analyzer, 4–10, 4–2, 4–3, 4–4
multiple actions for one RE, 4–10
null action, 4–10
yacc
ambiguous, 4–32
resolving, 4–34
conflicts, 4–32
resolving, 4–34
reduce, 4–31
shift, 4–31
yacc parsers, 4–18, 4–25
adding devices to the system, 8–46
address, sed editor, 3–4

admin command, 6–20, 6–27
ampersand
make, 7–11
sed, 3–12
angle brackets
lex, 4–5t
make, 7–15
archiving source files
See RCS
See SCCS
arithmetic, m4, 5–9
array, in awk, 2–13, 2–14
asterisk
in REs, 1–1t, 1–3
make, 7–10, 7–4
in $* macro, 7–15
at sign
make, 7–6
in $$@ macro, 7–14
in $@ macro, 7–14
awk
print command, 2–5
printf command, 2–5
awk command, 2–1 to 2–21
action
before or after processing the file, 2–8, 2–11, 2–3
omitting, 2–4
action operator, 2–14t
backslash, 2–8
BEGIN statement, 2–8
beginning of a field in an RE, 2–9
command-line syntax, 2–2
comments in programs, 2–19t
concatenating strings, 2–11
control structure, 2–19t, 2–18
awk command (cont.)
end of a field in an RE, 2–9
END statement, 2–8
field separator, 2–2t
field variable, 2–13
fields in, 2–1
function, 2–16t, 2–15
option, 2–2t
pattern, 2–3
omitting, 2–4
REs as patterns, 2–8
to specify ranges of records, 2–10
pipe, 2–20
program, 2–2
entering on the command line, 2–4e
syntax, 2–3
program structure, 2–4
ranges of records, 2–10
records in, 2–1
redirection, 2–20
relational expression, 2–9
REs as patterns, 2–8
semicolons in a program, 2–11n, 2–3
separating patterns from actions, 2–3
sequence of operations, 2–10, 2–4
slash, 2–8
split function, 2–13
string manipulation, 2–10, 2–11, 2–13
variable
array, 2–13, 2–14, 2–12, 2–6
built-in, 2–6t
creating, 2–12
field, 2–13
internal, 2–6t
RLENGTH, 2–16t
RSTART, 2–16t

awk command (cont.)
variable (cont.)
simple, 2–12
string, 2–12
treatment of, 2–12
value if uninitialized, 2–12

B

backslash
awk, 2–8
in REs, 1–1t
sed, 3–12, 3–9
Backus-Naur form, 8–31
backward link
See link, backward
BEGIN statement
awk, 2–8
lex, 4–15
BitTest shell function, 8–33
blank characters in macros, m4, 5–5
blank lines (spurious) in m4 output, 5–3
braces
awk, 2–3
lex, 4–14, 4–4
make, 7–9
yacc, 4–22
brackets
in REs, 1–1t
building programs
See lex program
See make utility
See yacc program
building software kits, 8–40
built-in macro
See macro

C

caret
See circumflex
changecom macro, m4, 5–8
changequote command, m4, 5–4
changequote macro, m4, 5–9
character class
in REs, 1–1t, 1–5
checking machine architecture, 8–32
ci command, 6–10, 6–13
circumflex
awk, 2–9
in REs, 1–1t
make, 7–10
collating sequence
in REs, 1–1t, 1–5
collating symbol
in REs, 1–5
colon, yacc, 4–24
comment characters, m4, 5–8
compression flag, 8–25t, 8–41
compression flag file, 8–41, 8–42
conditional action
m4, 5–12
make, 7–12
configuring software kits, 8–6
context address
sed, 3–5
control file
See kit, control file
control structure
awk, 2–19t, 2–18
sed, 3–10t
controlling revisions of source files
See RCS
See SCCS

create command, 6–20
creating a kit on distribution media
from a remote node, 8–45
on disk, 8–45
on tape, 8–45
prerequisites, 8–45
creating a new release
RCS, 6–13
SCCS, 6–24

D

data hierarchy, 8–21
declaration, yacc, 4–21
define command, m4, 5–2
defining macros
See m4 macro preprocessor
See make utility
deledit command, 6–25
delget command, 6–25
delta, 6–3
delta command, 6–24
dependency
between subsets, 8–30
checking, 8–32
file name expansion in, 8–31
for more than one version of a subset, 8–31
initializing for, 8–32
expression, 8–32
syntax and evaluation of, 8–31
locking, 8–30, 8–32, 8–4
establishing relationships, 8–32
initializating for, 8–32
unlocking, 8–33
initializating for, 8–32
dependency file
defined, 7–1
dependent
See dependency file
description file
make, 7–21e
command, 7–13, 7–5, 7–6, 7–9
testing, 7–20
diffs command, 6–26
directory
See also hierarchy
hierarchy
designing for kit installation, 8–7
standard
existence as links, 8–7
using for kit files, 8–7
layered products, 8–15
disktab file, 8–46
distributing kits across a network, 8–46
distribution media for kits, 8–45
disk, 8–45
tape, 8–45
divert macro, m4, 5–11
divnum macro, m4, 5–11
dlen macro, m4, 5–12
dnl command, m4, 5–3
dollar sign
awk, 2–9
in REs, 1–1t
m4, 5–5
make, 7–11, 7–4
sed, 3–5t, 3–4
dumpdef macro, m4, 5–13

E

edit command, 6–23
merging branches with, 6–26
–r option, 6–24
editing of files, simultaneous, management of by RCS, 6–5
editing of files, simultaneous, prevention of by SCCS, 6–6
egrep command
See grep command
embedded newline character
sed, 3–5t
end of file
lex, 4–14
sed, 3–4
endmarker token, 4–19, 4–24
value of, 4–23n
environment variable, in make, 7–15
error token, yacc, 4–29
escape character
in REs, 1–1
lex, 4–5t, 4–7
sed, 3–12
/etc/disktab file, 8–46
/etc/kitcap file, 8–45
eval macro, m4, 5–9
exporting kits, 8–17

F

fgrep command
See grep command
field
awk, 2–1, 2–13
file
attributes for setld kits, 8–21
file (cont.)
boot-required, in kits, 8–16
creating
RCS, 6–10
SCCS, 6–20
editing, SCCS, 6–23
getting multiple, SCCS, 6–24
getting status of, SCCS, 6–24
getting, SCCS, 6–22
layered product
linking to, 8–15, 8–19
physical location of, 8–16f, 8–15
lock, used by setld, 8–4
name in RCS, 6–11
name in SCCS, 6–20
naming for a product kit, 8–21n, 8–24
read-only, in kits, 8–16
read-write, in kits, 8–16
types of in a layered product, 8–16
used by setld, 8–3
versions in RCS, 6–3
versions in RCS or SCCS
identifying, 6–4
versions in SCCS, 6–3
file name
SCCS, 6–21
file system
guidelines for using in kit design, 8–7
links in, 8–7
standard hierarchy, 8–8f, 8–9t, 8–7
for layered products, 8–15
X hierarchy, 8–11f, 8–13t
finite-state automaton, 4–2f, 4–1, 4–30
stack usage, 4–30
flag
SCCS files, 6–22

flag (cont.)
SCCS files (cont.)
list of, 6–29t
sed, 3–12
forward link
See link, forward
function, in awk, 2–15

G

g-file, 6–3
gawk command
See awk command
gendisk command, 8–45
gentapes command, 8–45
get command, 6–23
–p option, 6–23
getting files from an RCS library, 6–12
specifying version, 6–12
getting files from an SCCS library, 6–22
for editing, 6–23
specifying version, 6–23
writing to standard output, 6–23
getting multiple SCCS files, 6–24
getting status of SCCS files, 6–24
grammar file, yacc, 4–21
contents of, 4–21
declarations section, 4–21
error, 4–28
guidelines, 4–27
programs section, 4–26
rules section, 4–24
grep command, 1–6 to 1–8
differences between grep, egrep, and fgrep, 1–6t
option, 1–7t

H

help command, 6–27
hierarchy
See also file system
data, for kit building, 8–21
output, for kit building, 8–40
required, for kit building, 8–18
source, for kit building, 8–19
populating, 8–20
target, for kit building, 8–19

I

ID keywords in SCCS, 6–21
See also percent sign
See also SCCS
ifdef macro, m4, 5–9
ifelse macro, m4, 5–12
image data file, 8–42
contents of, 8–42t
include macro, m4, 5–10
index macro, m4, 5–13
info command, 6–24
input/output routines, lex, 4–12
null character in, 4–13
overriding, 4–12
translation table for, 4–13
installing kits across a network, 8–46
installing software, 8–1 to 8–47
instctrl directory, 8–40
INSTCTRL file, 8–41
internal macro
See macro
inventory file
for kit building
See master inventory file

inventory file (cont.)
for a subset, 8–43
contents of, 8–44t

K

kernel, rebuilding for added devices, 8–46
key file, 8–24e, 8–24
contents of, 8–25t
control flag bit, 8–26t
using, 8–33
descriptor, 8–26t
keyword, processing, yacc, 4–22
associativity, 4–22
precedence, 4–22
kit
building, 8–40
adding a device for, 8–46
building process illustrated, 8–18f
configuring, 8–6
contents of, 8–3
control file, 8–3
contents of, 8–41t
location after installation, 8–33, 8–4
data hierarchy, 8–21
designing to simplify exporting, 8–17
distributing across a network, 8–46
installing, 8–5
installing across a network, 8–46
key file, 8–24
master inventory file, 8–21
output hierarchy, 8–40
planning locations of files, 8–7
removing, 8–7
SCP, 8–27
setting file attributes, 8–21
source hierarchy, 8–19
kit (cont.)
SPACE file, 8–45
standard directories, using for kit files, 8–7
layered products, 8–15
target hierarchy, 8–19
transferring to distribution media
from a remote node, 8–45
on disk, 8–45
on tape, 8–45
prerequisites, 8–45
using the file system effectively, 8–7
verifying, 8–6
kit, software, 8–1 to 8–47
kitcap file, 8–45
kits command, 8–40

L

layered product
boot-required files, 8–16
guidelines for placing files, 8–15
linking to files, 8–15, 8–19
physical location of files, 8–16f, 8–15
read-only files, 8–16
read-write files, 8–16
types of files in, 8–16
LC_TYPE environment variable
See collating sequence
len macro, m4, 5–12
lex analyzer
start condition, 4–15
setting, 4–15
lex library, 4–13, 4–3
lex program, 4–1 to 4–18
See also lexical analyzer
calculator example, 4–35
escape character, 4–5t, 4–7

lex program (cont.)
finding substrings, 4–9
matching wildcards, 4–8
quote characters, 4–5t, 4–7
REJECT action
alternative to, 4–12, 4–9
returning input to the input stream, 4–9
using, 4–15
using with yacc, 4–16
yyless function action, 4–12
lex utility
macro, 4–4
expansion, 4–4
substitution string, 4–4
lexical analyzer, 4–1 to 4–18
See also lex program
action, 4–10, 4–2, 4–3, 4–4
multiple for one RE, 4–10
null, 4–10
with yacc parsers, 4–10
action if no rule specified, 4–4
BEGIN statement, 4–15
default action, 4–4
end of file, 4–14
endmarker token, 4–19
file name, 4–16, 4–3
generating, 4–15
getting more input, 4–11
input look-ahead, 4–2
input/output routines, 4–12
null character in, 4–13
overriding, 4–12
translation table for, 4–13
length of a matched string, 4–11
lex library, 4–13, 4–3
passing code to generated program, 4–14
lexical analyzer (cont.)
printf function, 4–10
printing a matched string, 4–10
REs in, 4–5t, 4–3, 4–4, 4–5, 4–8
return statement, 4–17
returning input to the input stream, 4–12
extent of, 4–13
rule, 4–4
conflicts in, 4–8
matching input, 4–8
specification file
definitions section, 4–4
using y.tab.h in, 4–18
elements of, 4–3
format of, 4–3
incomplete, 4–5e
lines lex cannot interpret, 4–14
matching input, 4–8
rules section, 4–4
translation table, 4–13
yyleng variable, 4–11
yylval variable, 4–18
yymore function, 4–11
yytext variable, 4–10
yywrap function, 4–14
library, RCS
See RCS
library, SCCS
See SCCS
libscp function, 8–30t, 8–30
line number
sed, 3–4
link
backward, 8–36
creating, 8–40e, 8–39
initializating for, 8–39

link (cont.)
creating in SCPs, 8–36
forward, 8–36
literal string, yacc, 4–27
lock file, 8–4
subset dependency information in, 8–33, 8–4
locking subsets
See dependency, locking
look-ahead
lexical analyzer, 4–2
look-ahead token, yacc
clearing, 4–30
number, 4–19

M

m4 macro preprocessor, 5–1 to 5–13
arithmetic, 5–9
blank characters in macros, 5–5
changecom macro, 5–8
changequote macro, 5–9
conditional action, 5–12
defining macros, 5–2
terms of other macros, 5–3
to track other macros, 5–3
divert macro, 5–11
divnum macro, 5–11
dlen macro, 5–12
dnl macro, 5–3
dumpdef, 5–13
eval macro, 5–9
ifdef macro, 5–9
ifelse macro, 5–12
including a file, 5–10
index macro, 5–13
len macro, 5–12
macro
m4 macro preprocessor (cont.)
macro (cont.)
built-in, 5–6t
internal, 5–6t
macro argument, 5–5, 5–6
macro syntax, 5–1
maketemp macro, 5–11
print macro, 5–13
printing, 5–13
quote characters, 5–3
quoting in nested macros, 5–4
recursion, 5–2
redefining macros, 5–4
redirection, 5–11
spurious blank lines in output, 5–3
string manipulation, 5–12
substr macro, 5–12
temporary file, 5–11
translit macro, 5–13
undefine macro, 5–9
undivert macro, 5–11
using system programs, 5–11
machine architecture, checking for, 8–32
machine command, 8–32
macro
See also m4 macro preprocessor
See also make utility
argument, m4, 5–5, 5–6
built-in
m4, 5–6t
make, 7–13
checking for definition of, m4, 5–9
defined, m4, 5–1
defining
make, 7–8, 7–9
defining, m4, 5–2

macro (cont.)
defining, m4 (cont.)
terms of another macro, 5–3
to track another macro, 5–3
definition, in make, 7–3
expansion, m4
delaying, 5–4
recursive nature of, 5–2
internal
m4, 5–6t
make, 7–13t, 7–13
file name prefix, 7–15
out-of-date file list, 7–15
target file name, 7–14
first out-of-date file, 7–15
on dependency line, 7–14
lex, 4–4
expansion of, 4–4
nested, m4
quoting in, 5–4
precedence of definitions, in make, 7–3
redefining, m4, 5–4
removing, m4, 5–9
substitution, in make, 7–9
main function, yacc, 4–17, 4–18, 4–19
make utility, 7–1 to 7–22
command execution by, 7–3
command syntax, 7–3
conditional action, 7–12
creating files, 7–2
defining macros, 7–12, 7–8
dependency list, 7–5
description file, 7–21e, 7–3, 7–4, 7–9
example, 7–21, 7–7
environment variable, 7–15
including other files, 7–20
make utility (cont.)
internal macro, 7–13
file name prefix, 7–15
first out-of-date file, 7–15
out-of-date file list, 7–15
target file name, 7–14
on dependency line, 7–14
macro
internal, 7–13t
macro definition, 7–3
precedence, 7–3
macro substitution, 7–9
nested call, 7–13
on distributed system, 7–2
operation, 7–2
out-of-date file, 7–15, 7–2
recursion, 7–13
rule
defining, 7–17
internal, 7–16, 7–7
simplifying, 7–8
single suffix, 7–18
rules file example, 7–18e
shell invocation by, 7–3
suffixes
adding, 7–17, 7–16
replacing, 7–17
target file creation process, 7–2
target files with no dependents, 7–2
testing description files, 7–20
updating files, 7–2
using, 7–3
MAKEDEV command, 8–46
MAKEFLAGS macro, 7–15
maketemp macro, m4, 5–11

master inventory file, 8–22e, 8–21
contents of, 8–22t
creating and updating, 8–23
using pathnames in, 8–24
media supported by setld, 8–1, 8–46
merging branches of an SCCS file, 6–26
multiple matches in the sed editor, 3–12

N

\n
See embedded newline character
newinv utility, 8–23
NFS file sharing, 8–45
noninteractive editing
See sed editor
nonterminal symbol, 4–20, 4–22, 4–24
internal, 4–26
null character
grammar rule, 4–27
lex, 4–13
null string, yacc, 4–24

O

operator
action, in awk, 2–14t
Boolean, in awk, 2–10, 2–3
RE, defined for, 1–1
relational, in awk, 2–9
/opt directory, 8–15, 8–16
optional product
See layered product
output hierarchy, 8–40

P

p-file, 6–6
parentheses
awk, 2–3
in REs, 1–1t
m4, 5–10, 5–6
make, 7–9
parser, 4–18 to 4–39
See also yacc program
action, 4–25
ambiguous action, 4–32
resolving, 4–34
conflicting actions, 4–32
resolving, 4–34
controlling during a rule's action, 4–25
endmarker token, 4–19, 4–24
error handling, 4–28
to allow correction, 4–29
including the yylex function, 4–26
main function, 4–17, 4–19
reduce action, 4–31
shift action, 4–31
using with a lexical analyzer, 4–17
yychar variable, 4–19
yyerror function, 4–18, 4–19
yylex function, 4–18
yylval variable, 4–18
path
product inventories, 8–24
pattern
awk, 2–3, 2–4, 2–8
ranges of records, 2–10
pattern space, 3–3
percent sign
lex, 4–14, 4–4
SCCS, 6–21

percent sign (cont.)
yacc, 4–21, 4–22
period
in REs, 1–1t
pipes, in awk, 2–20
placing files in an RCS library, 6–10
placing files in an SCCS library, 6–20
plus sign
in REs, 1–1t, 1–3
postfix expression, 8–31
print command
in awk, 2–5
print macro, m4, 5–13
printf command
in awk, 2–5
processing text files
See awk command
See m4 macro preprocessor
See sed editor
product code, 8–25t
product name, 8–25t
product version, 8–25t
prs command, 6–27

Q

question mark
in REs, 1–1t, 1–4
make, 7–4
in $? macro, 7–15
quote characters, m4, 5–9
quoting strings
lex, 4–5t, 4–7
m4, 5–3

R

RCS, 6–1 to 6–34
ci command, 6–10, 6–13
creating a new release, 6–13
file names, 6–11
file storage, 6–3
getting files from the library, 6–12
specifying version, 6–12
ID keywords, 6–11t
library, 6–3
creating, 6–8
getting files from, 6–12
specifying version, 6–12
name of, 6–4
placing files in, 6–10
security, 6–8
placing files in the library, 6–10
preventing simultaneous editing of files, 6–5
rcsdiff command, 6–15
versions of files, 6–3
identifying, 6–4
rcs command
functions, 6–8t
RCS-file, 6–3
illustrated, 6–3f
rcsdiff command, 6–15
RE, 1–1 to 1–6
awk, 2–8
character classes, 1–5
collating considerations, 1–5
collating sequences, 1–5
concatenating multiple, 1–1
equivalence classes in, 1–6
escape character in, 1–1
internationalized usage, 1–5
length of attempted match, 1–3

RE (cont.)
lex, 4–5t, 4–3, 4–4, 4–5, 4–8
list of rules, for most utilities, 1–1
matching selected characters, 1–5
precedence of operators in, 1–3
restricting matches, 1–5
restricting matches in, 1–3, 1–4
rules, for most utilities, 1–1t
rules, for sed editor, 3–5t
saving and reusing patterns in, 1–4
specifying multiple, 1–5
record
awk, 2–1
recursion
m4, 5–2
make, 7–13
yacc, 4–28
redefining macros, m4, 5–4
redirection
awk, 2–20
m4, 5–11
reduce action, yacc, 4–31
regular expression
See RE
REJECT action, lex, 4–9
alternative to, 4–12
relational expression
awk, 2–9
release, creating new
RCS, 6–13
SCCS, 6–24
removing a macro, m4, 5–9
removing software kits, 8–7
repeating matches in the sed editor, 3–12
return statement, lex, 4–17
Revision Control System
See RCS
RLENGTH variable, in awk, 2–16t
RSTART variable, in awk, 2–16t
rule
lex, 4–4
conflicts in, 4–8
matching input, 4–8
make, 7–2
internal, 7–7
yacc, 4–24

S

s-file, 6–3
SCCS, 6–1 to 6–34
admin command, 6–20, 6–27
commands, 6–31t
create command, 6–20
creating a new release, 6–24
deledit command, 6–25
delget command, 6–25
delta command, 6–24
diffs command, 6–26
edit command, 6–23
merging branches with, 6–26
–r option, 6–24
file names, 6–20, 6–21
file storage, 6–3
g-file, 6–3
get command, 6–23
–p option, 6–23
getting files from the library, 6–22
for editing, 6–23
specifying version, 6–23
writing to standard output, 6–23
getting multiple files, 6–24

SCCS (cont.)
getting status of files, 6–24
help command, 6–27
ID keywords, 6–21t, 6–21
locating, 6–22
requiring, 6–22
info command, 6–24
library, 6–3
creating, 6–8
getting files from, 6–22
for editing, 6–23
specifying version, 6–23
writing to standard output, 6–23
name of, 6–4, 6–8
placing files in, 6–20
security, 6–31t, 6–8
specifying path to, 6–8
p-file, 6–6
placing files in the library, 6–20
preventing simultaneous editing of files, 6–6
prs command, 6–27
s-file, 6–3
sccsdiff command, 6–26
versions of files, 6–3
identifying, 6–4
sccs command, 6–18
–d option, 6–8
functions, 6–18t
options, list of, 6–31t
sccsdiff command, 6–26
SCP
See also dependency
aborting installation, 8–29
checking machine architecture, 8–32
creating links, 8–36
dependency function, 8–30t, 8–30
SCP (cont.)
example, 8–33
invoking, 8–28
language written in, 8–29
managing subset dependencies, 8–30
purpose, 8–27
restrictions on, 8–29
using key file control flag bits, 8–33
script
See sed editor, command script
searching for text
grep command, 1–6
security for RCS libraries, 6–8
security for SCCS libraries, 6–31t, 6–8
sed editor, 3–1 to 3–13
address, 3–4
limitations on using, 3–6
combining options, 3–2
command
buffer manipulation, 3–10t
editing, 3–7t
flow-of-control, 3–10t
command script, 3–1, 3–2
command syntax, 3–7e, 3–6
multiple, 3–3
on the command line, 3–2e
command-line syntax, 3–1
context address, 3–5
control structure, 3–10t
escape character, 3–12
hold area, 3–10t
input and output, 3–1
input file, treatment of, 3–2
limitations of, 3–1
line number, 3–4
option, 3–2t

sed editor (cont.)
order of operations, 3–3
pattern space, 3–3
printing lines
after substituting text, 3–13
repeating matches, 3–12
selecting lines for editing, 3–4
string manipulation, 3–12 to 3–13
substituting text
modifying command behavior, 3–12
using flags, 3–12
using an ampersand, 3–12
using backslashes, 3–12, 3–9
using semicolons, 3–2
using slashes, 3–5
using the hold area, 3–10t
writing a file, 3–13
semicolon
awk, 2–11n, 2–3
lex, 4–4
sed, 3–2
yacc, 4–24
setld
ACT environment variable, 8–28t, 8–28
media support, 8–1, 8–46
setld command, 8–1 to 8–47
aborting installation, 8–29
configuring kits, 8–6
files used, 8–3
installing kits with, 8–5
lock files used by, 8–4
option, 8–2t
descriptions of, 8–4 to 8–7
removing kits, 8–7
specifying an alternative root directory, 8–24n
setld command (cont.)
syntax and usage, 8–2
verifying kits, 8–6
shift action, yacc, 4–31
shift command, m4, 5–5
SID, 6–4
simultaneous editing of files, management of by RCS, 6–5
simultaneous editing of files, prevention of by SCCS, 6–6
sincldue macro, m4, 5–10
slash
awk, 2–8
sed, 3–7t, 3–5
software kit
See kit
Source Code Control System
See SCCS
source hierarchy, 8–19
populating, 8–20
SPACE file, 8–45
specification file, lex, 4–3
Definitions section, 4–4
using y.tab.h in, 4–18
format of, 4–3
incomplete, 4–5e
lines lex cannot interpret, 4–14
matching input, 4–8
rules section, 4–4
split function, in awk, 2–13
start condition, lex, 4–15
setting, 4–15
start symbol, yacc, 4–23
sticky bit
product kit, 8–26t

STL_ArchAssert shell function, 8–30t, 8–32
STL_DepEval shell function, 8–30t, 8–32
STL_DepInit shell function, 8–30t, 8–32
STL_DepLock shell function, 8–30t, 8–32
STL_DepUnLock shell function, 8–30t, 8–33
STL_LinkBack shell function, 8–39t, 8–39
STL_LinkInit shell function, 8–39t, 8–39
STL_LockInit shell function, 8–30t, 8–32
stream editor
See sed editor
string manipulation
awk, 2–10, 2–11, 2–13
lex, 4–4
m4, 5–12
sed, 3–12 to 3–13
string variable, in awk, 2–12
subset
compressing, 8–41
control flags in key file, 8–26t
using, 8–33
defined, 8–2
dependencies, 8–30
dependency locking, 8–30, 8–4
description in key file, 8–26t
descriptors in key file, 8–26t, 8–26
name, 8–26t
sticky bit, 8–26t
subset control file, 8–42
subset control program
See SCP
subset inventory file, 8–43
contents of, 8–44t
substr macro, m4, 5–12
substring, 4–9
symbol, yacc, 4–20, 4–22
start, 4–23
syntax
See individual utility entries
syscmd macro, m4, 5–11

T

target file
creation process in make, 7–2
defined, 7–1
without dependents, 7–2
target hierarchy, 8–19
temporary file, m4, 5–11
terminal symbol, 4–20
testing a bit in an integer, 8–33
time stamp
used by make utility, 7–2
token
m4
defined, 5–1
interpretation of, 5–2
yacc
defined, 4–16
finding names of, 4–17
list of, 4–20
token number, yacc, 4–23
translation table, lex, 4–13
translit macro, m4, 5–13

U

undefine macro, m4, 5–9
undivert macro, m4, 5–11
unlocking subset dependencies, 8–33
using make, 7–3
using the key file control flag bits, 8–33
/usr/opt directory, 8–15, 8–16

/usr/.smdb. directory, 8–33, 8–4

V

/var/opt directory, 8–15, 8–16
variable
awk
array, 2–13, 2–14, 2–12, 2–6
built-in, 2–6t
creating, 2–12
field, 2–13
internal, 2–6t
numeric, 2–12
simple, 2–12
treatment of, 2–12
value if uninitialized, 2–12
global, yacc, 4–22
verifying software kits, 8–6
vertical bar
in REs, 1–1t
lex, 4–10
yacc, 4–24

W

what command, 6–22

Y

y.tab.c file, 4–19
y.tab.h file, 4–35, 4–39
using in lex specification file, 4–18
yacc program, 4–16 to 4–39
See also parser
calculator example, 4–35
debug mode, 4–34
declaration, 4–21
finding token names, 4–17
yacc program (cont.)
global variable, 4–22
grammar file, 4–21
declarations section, 4–21
error, 4–28
guidelines, 4–27
programs section, 4–26
rules section, 4–24
library routines, 4–19
look-ahead token, clearing, 4–30
null character, 4–27
null string, 4–24
parameter keywords, 4–25
default values, 4–25
processing keywords
associativity, 4–22, 4–22
precedence, 4–22
recursion, 4–28
start symbol, 4–23
token number, 4–23
using with lex, 4–16
younger file
defined, 7–1
yy.lex.c file, 4–16, 4–3
yychar variable, 4–19
yyerror function, 4–18, 4–19
yyleng variable, 4–11
yyless function, lex, 4–12
yylex function, 4–18, 4–3
called by yyparse, 4–18
including in a parser, 4–26
requirements, 4–20
yylval variable, 4–18, 4–32
yymore function, 4–11
yyparse function, 4–18, 4–19

yytext variable, 4–10
yywrap function
lex, 4–14

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[a]	———	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Programming Support Tools
AA-PS32B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear - Fold Here ----

Cut
Along
Dotte
Line

Reader's Comments

DEC OSF/1
Programming Support Tools
AA-PS32B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- **Do Not Tear - Fold Here and Tape** ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- **Do Not Tear - Fold Here** ---

Cut Along Dotted Line