XENIX* 286
PROGRAMMER'S GUIDE

*XENIX is a trademark of Microsoft Corporation.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 174391-001

XENIX* 286
PROGRAMMER'S GUIDE

Order Number: 174391-001

*XENIX is a trademark of Microsoft Corporation

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

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

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BITBUS	i_m	iRMX	OpenNET
COMMputer	iMDDX	iSBC	Plug-A-Bubble
CREDIT	iMMX	iSBX	PROMPT
Data Pipeline	Insite	iSDM	Promware
Genius	int_el	iSXM	QUEST
$\hat{i}$	int_elBOS	KEPROM	QueX
i	Intelevision	Library Manager	Ripplemode
I^2ICE	$int_eligent$ Identifier	MCS	RMX/80
ICE	$int_eligent$ Programming	Megachassis	RUPI
iCS	Intellec	MICROMAINFRAME	Seamless
iDBP	Intellink	MULTIBUS	SLD
iDIS	iOSP	MULTICHANNEL	SYSTEM 2000
iLBX	iPDS	MULTIMODULE	UPI

XENIX is a trademark of Microsoft Corporation. Microsoft is a trademark of Microsoft Corporation. UNIX is a trademark of Bell Laboratories.

REV.	REVISION HISTORY	DATE
-001	Original issue	11/84

intel®

TABLE OF CONTENTS

CONTENTS

CHAPTER 1 INTRODUCTION	PAGE
Prerequisites	1-1
Manual Organization	1-1
Notation	1-2
CHAPTER 2 cc: C COMPILER	
Invoking the C Compiler	2-1
Creating Programs from C Source Files	2-2
Compiling a C Source File	3-2
Compiling Several Source Files	2-3
Naming the Output File	2-4
Creating Small, Middle, and Large Programs	2-4
Creating Small Model Programs	2-5
Creating Pure-Text Small Model Programs	2-6
Creating Middle Model Programs	2-6
Creating Large Model Programs	2-6
Using Object Files and Libraries	2-7
Creating Object Files	2-7
Creating Programs from Object Files	2-8
Linking a Program to Functions in Libraries	2-8
Creating Smaller and Faster Programs	2-9
Creating Optimized Object Files	2-9
Stripping the Symbol Table	2-9
Removing Stack Probes from a Program	2-10
Preparing Programs for Debugging	2-10
Producing an Assembly Language Listing	2-11
Profiling a Program	2-11
Controlling the C Preprocessor	2-12
Defining a Macro	2-12
Defining Include Directories	2-13
Ignoring the Default Include Directories	2-14
Saving a Preprocessed Source File	2-14
Error Messages	2-14
C Compiler Messages	2-15
Setting the Level of Warnings	2-15
Using Advanced Options	2-16
Creating Programs from Assembly Language Source Files	2-16
Using the near and far Keywords	2-17
Changing Word Order in Programs	2-18
Setting the Stack Size	2-18
Using Modules, Segments, and Groups	2-18

CONTENTS **PAGE**

Compiler Summary 2-20
cc Options 2-20
Memory Models 2-22
Pointer and Integer Sizes 2-22
Segment and Module Names 2-22

CHAPTER 3
lint: C PROGRAM CHECKER
Invoking lint 3-1
Checking for Unused Variables and Functions 3-2
Checking Local Variables 3-3
Checking for Unreachable Statements 3-4
Checking for Infinite Loops 3-5
Checking Function Return Values 3-5
Checking for Unused Return Values 3-6
Checking Types 3-6
Checking Type Casts 3-7
Checking for Nonportable Character Use 3-7
Checking for Assignment of longs to ints 3-8
Checking for Strange Constructions 3-8
Checking for Use of Older C Syntax 3-9
Checking Pointer Alignment 3-10
Checking Expression Evaluation Order 3-11
Embedding Directives 3-11
Checking for Library Compatibility 3-12

CHAPTER 4
make: PROGRAM MAINTAINER
Creating a Makefile 4-1
Invoking make 4-3
Using Pseudo-Target Names 4-4
Using Macros 4-5
Using Shell Environment Variables 4-7
Using the Built-In Rules 4-8
Changing the Built-In Rules 4-9
Using Libraries 4-11
Troubleshooting 4-12
Using make: An Example 4-13

CHAPTER 5
SCCS: SOURCE CODE CONTROL SYSTEM
Basic Information 5-1
Files and Directories 5-1
Deltas and SIDs 5-2
SCCS Working Files 5-3
SCCS Command Arguments 5-4
File Administrator 5-4
Creating and Using s-files 5-4
Creating an s-file 5-5
Retrieving a File for Reading 5-5
Retrieving a File for Editing 5-6

CONTENTS **PAGE**

Saving a New Version of a File 5-7
Retrieving a Specific Version 5-8
Changing the Release Number of a File 5-9
Creating a Branch Version 5-10
Retrieving a Branch Version 5-10
Retrieving the Most Recent Version 5-11
Displaying a Version 5-11
Saving a Copy of a New Version 5-12
Displaying Helpful Information 5-12
Using Identification Keywords 5-13
Inserting a Keyword into a File 5-13
Assigning Values to Keywords 5-14
Forcing Keywords 5-14
Using s-file Flags 5-14
Setting s-file Flags 5-15
Using the i Flag 5-15
Using the d Flag 5-15
Using the v Flag 5-16
Removing an s-file Flag 5-16
Modifying s-file Information 5-16
Adding Comments 5-17
Changing Comments 5-17
Adding Modification Requests 5-18
Changing Modification Requests 5-18
Adding Descriptive Text 5-19
Printing from an s-file 5-19
Using a Data Specification 5-19
Printing a Specific Version 5-20
Printing Later and Earlier Versions 5-20
Editing by Several Users 5-21
Editing Different Versions 5-21
Editing a Single Version 5-21
Saving a Specific Version 5-22
Protecting s-files 5-22
Adding a User to the User List 5-22
Removing a User from a User List 5-23
Setting the Floor Flag 5-23
Setting the Ceiling Flag 5-23
Locking a Version 5-24
Repairing SCCS Files 5-24
Checking an s-file 5-24
Editing an s-file 5-25
Changing an s-file's Checksum 5-25
Regenerating a g-file for Editing 5-25
Restoring a Damaged p-file 5-25
Using Other Command Options 5-26
Getting Help with SCCS Commands 5-26
Creating a File with the Standard Input 5-26
Starting at a Specific Release 5-26
Adding a Comment to the First Version 5-27
Suppressing Normal Output 5-27
Including and Excluding Deltas 5-28

CONTENTS **PAGE**

Listing the Deltas of a Version 5-29
Mapping Lines to Deltas 5-29
Naming Lines 5-29
Displaying a List of Differences 5-30
Displaying File Information 5-30
Removing a Delta 5-30
Searching for Strings 5-31
Comparing SCCS Files 5-31

CHAPTER 6
adb: PROGRAM DEBUGGER
Starting and Stopping adb 6-1
Starting with a Program File 6-1
Starting with a Core Image File 6-2
Starting adb with Data Files 6-3
Starting with the Write Option 6-3
Starting with the Prompt Option 6-3
Leaving adb 6-4
Displaying Instructions and Data 6-4
Forming Addresses 6-4
Forming Expressions 6-4
Decimal, Octal, and Hexadecimal Integers 6-5
Symbols 6-5
adb Variables 6-6
Current Address 6-7
Register Names 6-7
Operators 6-8
Choosing Data Formats 6-9
Using the = Command 6-10
Using the ? and / Commands 6-11
An Example: Simple Formatting 6-12
Debugging Program Execution 6-13
Executing a Program 6-13
Setting Breakpoints 6-14
Displaying Breakpoints 6-15
Continuing Execution 6-15
Stopping a Program with Interrupt and Quit 6-16
Single-Stepping a Program 6-16
Killing a Program 6-16
Deleting Breakpoints 6-17
Displaying the C Stack Backtrace 6-17
Displaying CPU Registers 6-18
Displaying External Variables 6-18
An Example: Tracing Multiple Functions 6-18
Using the adb Memory Maps 6-22
Displaying the Memory Maps 6-22
Changing the Memory Map 6-24
Creating New Map Entries 6-24
Validating Addresses 6-25
Miscellaneous Features 6-25
Combining Commands on a Single Line 6-25
Creating adb Scripts 6-26

CONTENTS **PAGE**

Setting Output Width 6-26
Setting the Maximum Offset 6-27
Setting Default Input Format 6-27
Using XENIX Commands 6-28
Computing Numbers and Displaying Text 6-28
An Example: Directory and Inode Dumps 6-29
Patching Binary Files 6-30
Locating Values in a File 6-30
Writing to a File 6-31
Making Changes to Memory 6-31

CHAPTER 7
as: ASSEMBLER
Command Usage 7-1
Lexical Conventions 7-2
Identifiers 7-2
Constants 7-2
White Space 7-2
Comments 7-2
Assembly Segments 7-3
Text, Data, and Bss Segments 7-3
The Location Counter 7-4
Statements 7-4
Labels 7-4
Null Statements 7-5
Expression Statements 7-5
Assignment Statements 7-5
Keyword Statements 7-6
Expressions 7-6
Expression Operators 7-6
Types 7-6
Type Propagation in Expressions 7-7
Assembler Directives 7-8
Even Directive 7-8
Floating-Point Directives 7-9
Global Directive 7-9
Segment Directives 7-9
Common Directive 7-10
Insert Directive 7-10
ASCII Directives 7-10
Listing Directives 7-11
Block Directives 7-11
Initial Value Directives 7-12
End Directive 7-12
Machine Instructions 7-12
Mnemonic List 7-12
Byte Instructions 7-16
Branch Instructions 7-17
String Instructions 7-17
Intersegment Instructions 7-18
Input/Output Instructions 7-18
80286 Instructions 7-18

CONTENTS **PAGE**

Addressing Modes 7-19
Register Operands 7-19
Immediate Operands 7-20
Direct Address Operands 7-20
Based Operands 7-21
Indexed Operands 7-21
Based Indexed Operands 7-22
Indirect Address Operands 7-22
Diagnostics 7-23

CHAPTER 8
csh: C SHELL
Invoking the C Shell 8-1
Using Shell Variables 8-2
Using the C Shell History List 8-4
Using Aliases 8-6
Redirecting Input and Output 8-7
Creating Background and Foreground Jobs 8-8
Using Built-In Commands 8-9
Creating Command Scripts 8-10
Using the argv Variable 8-11
Substituting Shell Variables 8-11
Using Expressions 8-13
Using the C Shell: A Sample Script 8-14
Using Other Control Structures 8-16
Supplying Input to Commands 8-17
Catching Interrupts 8-18
Using Other Features 8-18
Starting a Loop at a Terminal 8-19
Using Braces with Arguments 8-20
Substituting Commands 8-20
Special Characters 8-21

CHAPTER 9
lex: LEXICAL ANALYZER GENERATOR
lex Source Format 9-2
lex Regular Expressions 9-3
Invoking lex 9-4
Specifying Character Classes 9-5
Specifying an Arbitrary Character 9-6
Specifying Optional Expressions 9-6
Specifying Repeated Expressions 9-6
Specifying Alternation and Grouping 9-7
Specifying Context Sensitivity 9-7
Specifying Expression Repetition 9-8
Specifying Definitions 9-8
Specifying Actions 9-8
Handling Ambiguous Source Rules 9-12
Specifying Left Context Sensitivity 9-14
Specifying Source Definitions 9-17
lex and yacc 9-18
Specifying Character Sets 9-22
Source Format 9-22

CONTENTS **PAGE**

CHAPTER 10
yacc: COMPILER-COMPILER
Specifications 10-3
Actions 10-6
Lexical Analysis 10-8
How the Parser Works 10-9
Ambiguity and Conflicts 10-14
Precedence 10-18
Error Handling 10-21
The yacc Environment 10-23
Preparing Specifications 10-24
Input Style 10-24
Left Recursion 10-24
Lexical Tie-ins 10-25
Handling Reserved Words 10-26
Simulating Error and Accept in Actions 10-26
Accessing Values in Enclosing Rules 10-27
Supporting Arbitrary Value Types 10-27
A Small Desk Calculator 10-29
yacc Input Syntax 10-31
An Advanced Example 10-33
Old Features 10-39

CHAPTER 11
m4: MACRO PROCESSOR
Invoking m4 11-2
Defining Macros 11-2
Quoting 11-3
Using Arguments 11-5
Using Arithmetic Built-In Macros 11-6
Manipulating Files 11-7
Using System Commands 11-8
Using Conditionals 11-8
Manipulating Strings 11-9
Printing 11-10

APPENDIX A
C LANGUAGE PORTABILITY
Program Portability A-2
Machine Hardware A-2
Byte Length A-2
Word Length A-2
Storage Alignment A-3
Byte Order in a Word A-4
Bitfields A-5
Pointers A-5
Address Space A-6
Character Set A-6
Compiler Differences A-7
Signed/Unsigned char, Sign Extension A-7
Shift Operations A-7
Identifier Length A-7

CONTENTS **PAGE**

Register Variables A-8
Type Conversion A-8
Functions with Variable Number of Arguments A-9
Side Effects, Evaluation Order A-11
Program Environment Differences A-11
Portability of Data A-12
lint A-12
Byte Ordering Summary A-13

APPENDIX B
PROGRAMMING COMMANDS
adb B-3
admin B-12
ar B-17
as B-19
cb B-21
cc B-22
cdc B-27
comb B-30
config B-32
cref B-36
csh B-38
ctags B-58
delta B-60
get B-63
gets B-69
hdr B-70
help B-72
ld B-73
lex B-75
lint B-78
lorder B-81
m4 B-82
make B-86
mkstr B-93
nm B-95
prof B-97
prs B-98
ranlib B-102
ratfor B-103
regcmp B-105
rmdel B-106
sact B-107
sccsdiff B-108
size B-109
spline B-110
stackuse B-111
strings B-113
strip B-114
time B-116
tsort B-117
unget B-118
val B-119
xref B-121
xstr B-122
yacc B-124

CONTENTS **PAGE**

APPENDIX C
RELATED PUBLICATIONS

INDEX

TABLES

TABLE **TITLE** **PAGE**

2-1 Examples of near and far Keywords in a Small Model Program 2-17
2-2 Segments in Program Memory Models 2-22
2-3 Pointer and Integer Sizes in Program Memory Models 2-22
2-4 Default Segment and Module Names 2-22
A-1 Byte Ordering for Short Types A-13
A-2 Byte Ordering for Long Types A-13

intel®

CHAPTER 1
INTRODUCTION

This manual describes XENIX 286 Extended System commands used for developing and maintaining programs. These commands support program development, checking, debugging, maintenance, and version control.

Prerequisites

This manual presumes that you understand the C programming language and basic programming concepts. This manual also presumes some knowledge of XENIX or UNIX and the standard shell **sh**.

Manual Organization

This manual contains the following chapters and appendixes:

1. **Introduction:** manual overview, prerequisites, organization, and notation.

2. **cc: C Compiler:** the **cc** command for compiling C programs and assembling and linking the resulting modules.

3. **lint: C Program Checker:** a program that checks C programs for syntactic and semantic errors.

4. **make: Program Maintainer:** a program that automates the generation of program files, independent of the programming language or programming tools used.

5. **SCCS: Source Code Control System:** a set of commands to manipulate multiple versions of a single program or document, stored in a single file instead of in separate files for each version.

6. **adb: Program Debugger:** a program for debugging C or assembly language programs.

7. **as: Assembler:** the assembler used by **cc** to assemble compiled C programs.

8. **csh: C Shell:** a command interpreter that provides greater flexibility and more power than the standard shell **sh**.

9. **lex: Lexical Analyzer Generator:** a program that produces another program, called a lexical analyzer, that breaks up an input stream into "tokens," using rules specified to the generator.

10. **yacc: Compiler-Compiler:** a program that produces a parser program that translates input according to rules specified to **yacc.**

11. **m4: Macro Processor:** a program that processes macros defined and used in its input, producing output with the macros replaced by the text that they stand for.

A. **C Language Portability:** how to write C language programs so that they are portable to other XENIX systems running on other types of hardware.

B. **Programming Commands:** reference information for all the programming commands in the XENIX 286 Extended System.

C. **Related Publications:** descriptions and ordering information for all XENIX 286 Release 3 manuals and any other publications referenced by this manual.

Notation

These notational conventions are used in this manual:

- Literal names are bolded where they occur in text, e.g., **/sys/include, printf, dev_tab, EOF.**
- Syntactic categories are italicized where they occur and indicate that you must substitute an instance of the category, e.g., *filename.*
- In examples of dialogue with the XENIX 286 system, characters entered by the user are bolded.
- In syntax descriptions, optional items are enclosed in brackets (**[ ]**).
- Items that can be repeated one or more times are followed by an ellipsis (**...**).
- Items that can be repeated zero or more times are enclosed in brackets and followed by an ellipsis (**[ ]...**).
- A choice between items is indicated by separating the items with vertical bars (**|**).

intel®

CHAPTER 2
cc: C COMPILER

This chapter explains how to use the **cc** command. In particular, it explains how to

- Compile C language source files
- Choose a memory model for a program
- Use object files and libraries with a program
- Create smaller and faster programs
- Prepare C programs for debugging
- Control the C preprocessor

It also describes the error and warning messages generated by the C compiler and explains how to use advanced features of the **cc** command to make customized programs.

This chapter assumes that you are familiar with the C programming language and that you can create C program source files using a XENIX text editor.

Invoking the C Compiler

The **cc** command has the form

cc [*option*] ... *filename* ...

where *option* is a command option, and *filename* is the name of a C language source file, an assembly language source file, an object file, or an archive library. You may give more than one option or file name, if desired, but you must separate each item with one or more spaces.

The **cc** command options control and modify the tasks performed by the command. For example, you can direct **cc** to perform optimization or create an assembly listing file. The options also specify additional information about the compilation, such as which program libraries to examine and what the name of the executable file should be. Many options are described in the following sections. For a complete description of all options, see the entry **cc** in Appendix B, "Programming Commands."

Creating Programs from C Source Files

The **cc** command is normally used to create executable programs from C language source files. A file's contents are identified by the file name extension. C source files must have the extension ".c".

The **cc** command can create executable programs only from source files that make up a complete C program. In XENIX, a complete program must have one (and only one) function named "main". This function becomes the entry point for program execution. The "main" function may call other functions as long as they are defined within the program or are part of the C standard library. The standard C library is described in the *XENIX 286 C Library Guide*.

Compiling a C Source File

You can compile a C source file by giving the name of the file when you invoke the **cc** command. The command compiles the statements in the file, then copies the executable program to the default output file **a.out**.

To compile a source program, type

```
cc filename
```

where *filename* is the name of the file containing the program. The program must be complete, that is, it must contain a "main" program function. It may also contain calls to functions explicitly defined by the program or by the standard C library.

For example, assume that the following program is stored in the file named **main.c**.

```
#include <stdio.h>

main ()
{
        int x,y;

        scanf("%d %d", &x, &y);
        printf("%d\n", x + y);
}
```

To compile this program, type

```
cc main.c
```

The command first invokes the C preprocessor, which adds the statements in the file **/usr/include/stdio.h** to the beginning of the program. It then compiles these statements and the rest of the program statements. Next, the command links the program with the standard C library, which contains the object files for the **scanf** and **printf** functions. Finally, it copies the program to the file **a.out**.

You can execute the new program by typing

```
a.out
```

The program waits until you enter two numbers, then prints their sum. For example, if you type "3 5" the program displays "8".

Compiling Several Source Files

Large source programs are often split into several files to make them easier to understand, update, and edit. You can compile such a program by giving the names of all the files belonging to the program when you invoke the **cc** command. The command reads and compiles each file in turn, then links all object files together, and copies the new program to the file **a.out.**

To compile several source files, type

```
cc filename ...
```

where each *filename* is separated from the next by at least one space. One of these files (and only one) must contain a "main" function. The others may contain functions called by this "main" function or by other functions in the program. The files must not contain calls to functions not explicitly defined by the program or by the standard C library.

For example, suppose the following main program function is stored in the file **main.**

```
#include <stdio.h>
extern int add();

main ()
{
        int x,y,z;

        scanf ("%d %d", &x, &y);
        z = add (x, y);
        printf ("%d \n", z);
}
```

Assume that the following function is stored in the file **add.c.**

```
add (a, b)
int a, b;
{
        return (a + b);
}
```

You can compile these files and create an executable program by typing

```
cc  main.c  add.c
```

The command compiles the statements in **main.c,** then compiles the statements in **add.c.** Finally, it links the two together (along with the standard C library) and copies the program to **a.out.** This program, like the program in the previous section, waits for two numbers, then prints their sum.

Since the **cc** command cannot keep track of more than one compiled file at a time, when several source files are compiled at a time, the command creates object files to hold the binary code generated for each source file. These object files are then linked to create an executable program. The object files have the same base name as the source files but are given the ".o" file extension. For example, when you compile the two source files above, the compiler produces the object files **main.o** and **add.o.** These files are permanent files, i.e., the command does not delete them after completing its operation. Note that the command also creates an object file if only one source file is compiled.

Naming the Output File

You can give the executable program file any valid file name by using the **-o** (for "output") option. The option has the form

```
-o filename
```

where *filename* is a valid file name or path name. If a file name is given, the program file is stored in the current directory. If a full path name is given, the file is stored in the given directory. If that file already exists, its contents are replaced with the new executable program.

For example, the command

```
cc -o addem main.c add.o
```

causes the compiler to create an executable program file **addem** from the source file **main.c** and object file **add.o.** You can execute this program by typing

```
addem
```

Note that the **-o** option does not affect the existing **a.out** file. This means that the **cc** command does not change the current contents of **a.out** if the **-o** option has been given.

Creating Small, Middle, and Large Programs

The **cc** command supports the creation of programs of a variety of sizes and purposes using the **-Ms, -Mm, -Ml,** and **-i** options. These options define the size of a given program by defining the number of segments in physical memory to be allocated for your program's use. They also determine how the system loads the program for execution.

The **cc** command allows the creation of programs in four different memory models: impure-text small model, pure-text small model, middle model, and large model. Each model defines a different type of program structure and storage.

Impure-text small model programs are typically C programs that are short or have a limited purpose. These programs must not exceed 64 Kbytes.

Pure-text small model programs are typically short programs intended to be invoked by many users. Pure-text programs can occupy up to 128 Kbytes, but no more than 64 Kbytes each is permitted for either instructions or data. Unlike impure-text small model programs, the system loads only one copy of a pure-text program's instructions into memory, no matter how many times it has been invoked. As long as this copy stays in memory, the system simply loads a new copy of the data for each new invocation of the program. It then keeps each copy of data separate, while sharing the instructions among the different invocations. Pure-text programs save valuable memory space that would otherwise be wasted by impure-text small model programs.

Middle model programs are typically C programs that have a large number of program statements but a relatively small amount of data. Program instructions can be any size, but program data must not exceed 64 Kbytes.

Large model programs are typically very large C programs that use a large amount of data storage during normal processing. Program instructions and data may have any size, except that the program must not contain arrays or structures that exceed 64 Kbytes.

C programs in memory consist of the actual machine instructions created from the program's source statements, and the binary data storage created for the program's variables. The data storage also contains the stack used by the program for temporary storage during execution. The XENIX system stores the instructions and data in one or more segments of physical memory. Each segment is 64 Kbytes long. Thus, the maximum allowable size for any program depends on how many segments are allocated for it when compiled.

The following sections describe how to use the **-M** and **-i** options to create programs with a specific number of segments. They also describe how to create pure-text programs for execution by multiple users.

Creating Small Model Programs

You can create a small model program by using the **-Ms** option. This option directs **cc** to create a program that occupies a single segment when loaded into physical memory. To create a small model program, type

```
cc -Ms filename
```

where *filename* is the name of the program you wish to compile.

The **cc** command creates small model programs by default when you do not otherwise specify a program model. Thus, the **-Ms** option is not required.

Creating Pure-Text Small Model Programs

You can create a pure-text small model program by combining the **-i** and **-Ms** options. The **-i** option directs **cc** to create separate memory segments for the instructions and data of a small model program. To create a pure-text program, type

```
cc -Ms -i filename
```

where *filename* is the name of the file source program to be compiled. Since **cc** creates small model programs by default, only the **-i** option is required.

Creating Middle Model Programs

You can create a middle model program by using the **-Mm** option. This option creates one segment for the data of the program and one or more segments for the instructions. To create a middle model program, type

```
cc -Mm filename
```

where *filename* is the name of the source file to be compiled. When creating a program, the compiler attempts to fit as many instructions into a segment (up to 64 Kbytes) as possible. If the program is larger than 64 Kbytes, the **-NGT** option must be used to create new segments (see the section "Using Modules, Segments, and Groups" later in this chapter).

Middle model programs are pure in the sense that the system never loads more than one copy of the program's instructions into memory at one time. Thus the **-i** option, used with pure-text small model programs, is not required for middle model programs.

Creating Large Model Programs

You can create large model programs by using the **-Ml** option. This option directs **cc** to create multiple segments for both instructions and data. To create a large model program, type

```
cc -Ml filename
```

where *filename* is the name of a source file to be compiled.

As with middle model programs, the compiler attempts to fit as many instructions into a segment as possible. If a program's instructions or data is greater than 64 Kbytes each, the **-NGT** and **-NGD** options must be used to create new segments (see the section "Using Modules, Segments, and Groups" later in this chapter).

Like middle model programs, large model programs are considered to be pure.

Using Object Files and Libraries

The **cc** command can save useful functions as object files and use these object files to create programs at a later time. Object files contain the compiled or assembled instructions of your source file, so they save you the time and trouble of recompiling the functions each time you need them. All object files created by **cc** have the file extension ".o".

The **cc** command can also use functions found in XENIX system libraries, such as the standard C library or the screen processing library **curses.** To use these functions, you simply supply the name of the library containing them. In some cases, such as for the standard C library, **cc** accesses the library automatically and no explicit naming is required.

For convenience, you can create your own libraries with the **ar** and **ranlib** commands. These commands, described in Appendix B, "Programming Commands," copy your useful object files to a library file and prepare the file for use by the **cc** command. You can access the library like any other library in the system if you copy it to the **/lib** directory.

Creating Object Files

You can create an object file from a given source file by using the **-c** (for "compile") option. This option directs **cc** to compile the source file without creating a final program. The option has the form

```
-c filename ...
```

where *filename* is the name of the source file. You may give more than one file name if you wish. Make sure each name is separated from the next by a space.

To make object files for the source files **add.c** and **mult.c,** type

```
cc -c add.c mult.c
```

This command compiles each file and copies the compiled source files to the object files **add.o** and **mult.o.** It does not link these files; no executable program is created.

The **-c** option is typically used to save useful functions for programs to be developed later. Once a function is in an object file it may be used as is, or saved in a library file and accessed like other library functions, as described in the following sections.

Note that the **cc** command automatically creates an object file for each source file in the command line. Unless the **-c** option is given, however, it will also attempt to link these files, even if they do not form a complete program.

Creating Programs from Object Files

You can use the **cc** command to create executable programs from one or more object files, or from a combination of object files and C source files. The command compiles the source files (if any), then links the compiled source files with the object files to create an executable program.

To create a program, give the names of the object and source files you wish to use. For example, if the source file **main.c** contains calls to the functions **add** and **mult** (saved in the object files **add.o** and **mult.o**), you can create a program by typing

```
cc main.c add.o mult.o
```

In this case, **main.c** is compiled and then linked with **add.o** and **mult.o** to create the executable file **a.out.**

Linking a Program to Functions in Libraries

You can link a program to functions in a library by using the -l (for "library") option. The option directs **cc** to search the given library for the functions called in the source file. If the functions are found, the command links them to the program file.

The option has the form

```
cc -lname
```

where *name* is a shortened version of the library's actual file name (see the *XENIX 286 C Library Guide* for a list of names). Spaces between the name and option are optional. The linker searches the **/lib** directory for the library. If not found, it searches the **/usr/lib** directory.

For example, the command

```
cc main.c -lcurses
```

links the library **/lib/libcurses.a** to the source file **main.c.**

A library is a convenient way to store a large collection of object files. The XENIX system provides several libraries, the most common of which is the standard C library. Functions in this library are automatically linked to your program whenever you invoke the compiler. Other libraries, such as **libcurses.a,** must be explicitly linked using the -l option. The XENIX libraries and their functions are described in detail in the *XENIX 286 C Library Guide.*

In general, the **cc** command does not search a library until the -l option is encountered, so the placement of the option is important. The option must follow the names of any source files containing calls to functions in the given library. In general, all library options should be placed at the end of the command line, after all source and object files.

Creating Smaller and Faster Programs

You can create smaller and faster C programs by using the optimizing options available with the **cc** command. These options reduce the size of a compiled program by removing unnecessary or redundant instructions or unnecessary symbol information. Smaller programs usually run faster and save valuable space.

Creating Optimized Object Files

You can create an optimized object file or an optimized program from a given source file by using the **-O** (for "optimize") option. This option reduces the size of the object file or program by removing unnecessary instructions. For example, the command

```
cc -O main.c
```

creates an optimized program from the source file **main.c.** The resulting object file or program is smaller (in bytes) than if the source had been compiled without the option. A smaller object file usually means faster execution.

The **-O** option applies to source files only; existing object files are ignored if included with this option. The option must appear before the names of the files you wish to optimize. For example, the command

```
cc -O add.c main.c
```

optimizes **add.c** and **main.c.**

You can combine the **-O** and **-c** options to compile and optimize source files without linking the resulting object files. For example, the command

```
cc -O -c main.c add.c
```

creates separate optimized object files from the source files **main.c** and **add.c.**

Although optimization is very useful for large programs, it takes more time than regular compilation. In general, it should be used in the last stage of program development, after the program has been debugged.

Stripping the Symbol Table

You can reduce the size of a program's executable file by using the **-s** and **-x** options. These options direct **cc** to remove items from the symbol table. The symbol table contains information about code relocation and program symbols and is used by the XENIX debugger **adb** to allow symbolic references to variables and functions when debugging. The information in this table is not required for normal execution and should be removed when the program has been completely debugged.

The **-s** option strips the entire table, leaving machine instructions only. For example, the command

```
cc -s main.c add.c
```

creates an executable program that contains no symbol table. It also creates the object files **main.o** and **add.o,** which contain no symbol tables.

The **-x** option strips all nonglobal symbols from the file including the names of local functions and variables, but excluding externally declared items. The command

```
cc -x main.o add.o
```

creates an executable program with global symbols, but only if the object files **main.o** and **add.o** have symbol tables.

The **-s** and **-x** options may be combined with the **-O** option to create an optimized and stripped program. Note that you can also strip a program with the XENIX command **strip,** described in Appendix B, "Programming Commands."

Removing Stack Probes from a Program

You can reduce the size of a program slightly by using the **-K** option to remove all stack probes. A stack probe is a short routine called by a function to check the program stack for available space. The probes are not needed if the program makes very few function calls or has unlimited stack space.

To remove the stack probes from the program **main.c,** type

```
cc -K main.c
```

Although this option, when combined with the **-O** and **-s** options, makes the smallest possible program, it should be used with great care. Removing stack probes from a program with stack use that is not well known can cause execution errors.

Preparing Programs for Debugging

The **cc** command provides a variety of options to prepare a program for debugging. These options range from creating an assembly language listing of the program for use with the XENIX debugger **adb,** to adding routines for profiling the execution of a program.

Producing an Assembly Language Listing

You can direct the compiler to generate an assembly language listing of your compiled source file by using the **-S** and **-L** options. The **-S** option creates an assembly language listing that can later be assembled by resubmitting it (perhaps after revision) to **cc.** The **-L** option creates a listing that shows assembled code, as well as instructions. The file created by **-S** is given the file extension ".s"; the file created by **-L** is given ".L".

Assembly language listing files are typically used by programmers who wish to debug their program with **adb.** Since **adb** recognizes machine instructions instead of the actual source statements in a program, a programmer needs an assembly language listing for accurate debugging.

To create an assembly language listing, give the name of the desired source file. For example, the command

```
cc -S add.c
```

creates an assembly language listing file named **add.s** and the command

```
cc -L mult.c
```

creates a listing file named **mult.L.** Note that both the **-S** and **-L** options suppress subsequent compilation of the source file; they imply the **-c** option. Thus, no program file is created and no linking is performed.

Another use of the **-S** option is to create an assembly language source file that may be optimized by hand and later submitted to **cc** for translation (see "Creating Programs from Assembly Language Source Files," below). Although this method can be useful, optimizing should be left to the compiler whenever possible.

The **-S** and **-L** options apply to source files only; the compiler cannot create an assembly language listing file from an existing object file. Furthermore, the options in the command line must appear before the names of the files for which the assembly listings are to be saved.

Profiling a Program

You can examine the flow of execution of a program by adding "profiling" code to the program with the **-p** option. The profiling code automatically keeps a record of the number of times program functions are called during execution of the program. This record is written to the **mon.out** file and can be examined with the **prof** command.

For example, the command

```
cc -p main.c
```

adds profiling code to the program created from the source file **main.c.** The profiling code automatically calls the **monitor** function, which creates the **mon.out** file at normal termination of the program. The **prof** command is described in Appendix B, "Programming Commands." The **monitor** function is described in "System Functions" in the *XENIX 286 C Library Guide.*

The **-p** option must be given in any command line that references object files containing profiling code. For example, if the command

```
cc -c -p f1.c f2.c
```

was used to create the object files **f1.o** and **f2.o,** then the command

```
cc -p f1.o f2.o
```

must be used to create an executable program from these files.

Controlling the C Preprocessor

The **cc** command provides a number of options to control the operation of the C preprocessor. These options can define macros, create new search paths for include files, and suppress subsequent compilation of the source file.

Defining a Macro

You can define the value or meaning of a macro used in a source file by using the **-D** (for "define") option. The **-D** option assigns a value to a macro when you invoke the compiler and is useful if you have used **if, ifdef,** and **ifndef** directives in your source files.

The option has the form

-D*name*[= *string*]

where *name* is the name of the macro and *string* is its value or meaning. If no *string* is given, the macro is assumed to be defined and its value is set to 1. For example, the command

```
cc -DNEED=2 main.c
```

sets the macro "NEED" to the value "2". This is the same as having the directive

```
#define NEED 2
```

in the source file. The command compiles the source file **main.c,** replacing every occurrence of "NEED" with "2".

The **-D** option is especially useful with the **ifdef** directive. You can use the option to determine which statements in the source are to be compiled. For example, suppose a source file, **main.c,** contains the directive

```
#ifdef NEED
```

but no explicit **define** directive for the macro "NEED". Then all statements following the **ifdef** directive are compiled only if you supply an explicit definition of "NEED" by using **-D.** For example, the command

```
cc -DNEED main.c
```

is sufficient to compile all statements following the **ifdef** directive, while the command

```
cc main.c
```

causes all those statements to be ignored.

You may use **-D** to define up to 20 macros on a command line. However, you cannot redefine a macro once it has been defined. If a file uses a macro, you must place the **-D** option before that file's name on the command line. For example, in the command

```
cc main.c -DNEED a dd.c
```

the macro "NEED" is defined for **add.c** but not defined for **main.c.**

Defining Include Directories

You can explicitly define the directories containing "include" files by using the **-I** (for "include") option. This option adds the given directory to a list of directories to be searched for include files. The directories in the list are searched whenever an include directive is encountered in the source file. The option has the form

```
-Idirectoryname
```

where *directoryname* is a valid path name to a directory containing include files. For example, the command

```
cc -I/usr/joe/include main.c
```

causes the compiler to search the directory **/usr/joe/include** for include files requested by the source file **main.c.**

The directories are searched in the order they are listed and only until the given include file is found. The **/usr/include** directory is the default include directory and is always searched after directories given with **-I.**

Ignoring the Default Include Directories

You can prevent the C preprocessor from searching the default include directories by using the **-X** option. This option is generally used with the **-I** option to define the location of include files that have the same names as those found in the default directories, but which contain different definitions. For example, the command

```
cc -X -I/usr/joe/include main.c add.c
```

causes **cc** to look for all include files only in the directory **/usr/joe/include.**

Saving a Preprocessed Source File

You can save a copy of the preprocessed source file by using the **-P** and **-E** options. The file is identical to the original source file except that all C preprocesor directives have been expanded or replaced. The **-P** option copies the result to the file named *filename*.**i,** where *filename* is the same name as the source file without the ".c" extension. The **-E** option copies the result to the standard output and places a **<ine** directive at the beginning and end of this output. You can save this output by redirecting it.

For example, the command

```
cc -P main.c
```

creates a preprocessed file **main.i** from the source file **main.c,** and the command

```
cc -E add.c >add.i
```

creates a preprocessed file from the source file **add.c.** The output is redirected to the file **add.i.**

Note that **-P** and **-E** suppress compilation of the source file. Thus, no object file or program is created.

Error Messages

The C compiler generates a broad range of error and warning messages to help you locate errors and potential problems in programs. In addition to compiler messages, the **cc** command also displays error messages generated by the XENIX C preprocessor and the XENIX assembler and linker programs. The following sections describe the form and meaning of the compiler error messages and warning messages you can encounter while using the **cc** command.

C Compiler Messages

The C compiler displays messages about syntactic and semantic errors in a source file, such as misplaced punctuation, illegal use of operators, and undeclared variables. It also displays warning messages about statements containing potential problems caused by data conversions or the mismatch of types. Error and warning messages have the form

filename (*linenumber*) : *message*

where *filename* is the name of the source file being compiled, *linenumber* is the number of the line in the source file containing the error, and *message* is a self-explanatory description of the error or warning.

If an error is severe, the compiler displays a message and terminates the compilation. Otherwise, the compiler continues looking for other errors but does not create an object file. If only warning messages are displayed, the compiler completes compilation and creates an object file.

You can avoid many C compiler errors by using the XENIX C program checker **lint** before compiling your C source files. **lint** performs detailed error checking on a source file and provides a list of actual errors and possible problems that may affect execution of the program. For a description of **lint**, see Chapter 3, **"lint:** C Program Checker."

Setting the Level of Warnings

You can set the level of warning messages produced by the compiler by using the **-W** option. This option directs the compiler to display messages about statements that may not be compiled as the programmer intends. Warnings indicate potential problems rather than actual errors. The option has the form

-W *number*

where *number* is a number in the range 0 to 3 giving the level of warnings. The levels are

0 Suppresses all warning messages. Only messages about actual syntactic or semantic errors are displayed.

1 Warns about potentially missing statements, unreachable statements, and other structural problems. Also warns about overt type mismatches.

2 Warns about all type mismatches (strong typing).

3 Warns on all automatic data conversions.

If the option is not used, the default is level 1.

The higher option levels are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower levels are best for compiling programs whose questionable statements are intentionally designed. For example, the command

```
cc -W 3 main.c
```

directs the compiler to perform the highest level of checking and produces the greatest number of warning messages. The command

```
cc -W 0 main.c
```

produces no warning messages. Note that the **-w** option has the same effect as **-W 0.**

Using Advanced Options

The **cc** command provides a number of advanced programming options that give greater control over the compilation process and the final form of the executable program. The following sections describe a number of these options.

Creating Programs from Assembly Language Source Files

You can use the **cc** command to create executable programs from a combination of C source files and 8086/286 assembly language source files. Assembly language source files must contain 8086/286 instructions, as described in Chapter 7, "**as:** Assembler," and must have the extension ".s".

When assembly language source files are given, the **cc** command uses its own internal assembler to assemble the instructions and create an object file. The object file can then be linked with object files created by the compiler. For example, the command

```
cc main.c add.s
```

compiles the C source file **main.c** but assembles the assembly language source file **add.s.** The resulting object files, **main.o** and **add.o,** are linked to form a single program.

Assembly language files produced by **cc** using the **-S** option can be assembled by **cc** but *cannot* be assembled by the assembler program **as.** Similarly, assembly language files in the format used by **as** cannot be assembled by **cc.**

When using assembly language routines with C programs, you must be sure to provide the correct interface for calls to and from C language functions. C functions require a specific calling and return sequence. Assembly language functions that fail to provide this interface will cause errors. See "Assembly Language Interface" in the *XENIX 286 C Library Guide.*

Using the near and far Keywords

The **near** and **far** keywords are special type modifiers that define the length and meaning of the address of a given variable. The **near** keyword defines an object with a 16-bit address. The **far** keyword defines an object with a full 32-bit segmented address. Any data item or function can be accessed.

The **near** and **far** keywords override the normal address length generated by the compiler for variables and functions. In small model programs, **far** can be used to access data and functions in segments outside of the program. In middle and large model programs, **near** can be used to access data with just an offset.

The examples in Table 2-1 illustrate the **far** and **near** keywords as used in declarations in a small model program.

Table 2-1. Examples of **near** and **far** Keywords in a Small Model Program

Declaration	Address Size	Item Size
char c;	near (16 bits)	8 bits (data)
char far d;	far (32 bits)	8 bits (data)
char *p;	near (16 bits)	16 bits (near pointer)
char far *q;	near (16 bits)	32 bits (far pointer)
char * far r;	far (32 bits)	16 bits (near pointer)*
char far * far s;	far (32 bits)	32 bits (far pointer)**
int foo();	near (16 bits)	function returning 16 bits
int far foo();	far (32 bits)	function returning 16 bits***

*This example has no meaning; it is shown for syntactic completeness only.

**This is similar to accessing data in a long model program.

***This example leads to trouble in most environments. The far call changes the CS register and makes run time support unavailable.

The following example is from a middle model compilation:

```
int near foo();
```

This does a near call in an otherwise far (calling) program.

Since there is no type checking between items in separate source files, the **near** and **far** keywords should be used with great care.

Changing Word Order in Programs

The C compiler automatically uses the standard 8086/286 word order for **long** type values. This order may cause problems when programs access data files generated by programs created by other C compilers. You can change the word order for a given program by using the **-Mb** configuration option. This option causes the compiler to generate all long values in reverse word order, making the program compatible with programs created by other XENIX compilers.

Other portability issues must be considered when creating C programs intended for different computer systems. For an explanation of these issues, see Appendix A, "C Language Portability."

Setting the Stack Size

You can set the size of the program stack by using the **-F** option. This option has the form

-F *num*

where *num* is the size (in bytes) of the program stack. The program stack is used for storage of function parameters and automatic variables. If the option is not used, a default stack size (4,096 bytes) is set.

You can determine the stack requirements of a given program by using the **stackuse** command, described in Appendix B. This command analyzes C source files and computes the minimum stack requirement for all functions in the program. The command displays a warning if recursive functions are encountered; stack use requirements for recursive functions must be determined by the programmer.

Note that all programs created by **cc** have fixed stacks. This means the stack size cannot be increased during execution of the program. Therefore, a sufficient stack size must be given when compiling the program.

Using Modules, Segments, and Groups

"Module" is another name for the object file created by the C compiler. Every module has a name, and the **cc** command uses this name in error messages if problems are encountered during linking. The module name is usually the same as the source file's name (without the ".c" or ".s" extension). You can change this name using the **-NM** option. The option has the form

-NM *name*

where *name* can be any combination of letters and digits.

Changing a module's name is useful if the source file to be compiled is actually the output of a program preprocessor and generator, such as **lex** or **yacc.**

A "segment" is a contiguous block of binary code produced by the C compiler. Every module has two segments: a text segment containing the program instructions, and a data segment containing the program data. Each segment in every module has a name. This name is used by **cc** to define the order in which the segments of the program will appear in memory when loaded for execution. Text segments having the same name are loaded as a contiguous block of code. Data segments of the same name are also loaded as contiguous blocks.

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small model programs the text segment is named "_TEXT" and the data segment is named "_DATA". These names are the same for all small model modules, so all segments from all modules of a small model program are loaded as a contiguous block. In middle model programs, each text segment has a different name. In large model programs, each text and data segment has a different name. The default text and data segment names for middle and large model programs are given in the section "Segment and Module Names" at the end of this chapter.

You can override the default names used by the C compiler (and override the default loading order) by using the **-NT** and **-ND** options. These options set the names of the text and data segments, in each module being compiled, to a given name. The options have the form

-NT *name*

and

-ND *name*

where *name* is any combination of letters and digits. These options are useful in middle and large model programs where there is no specific loading order. In these programs, you can guarantee contiguous loading for two or more segments by giving them the same name.

All text and data segments, whether or not they are loaded as contiguous blocks, are eventually loaded into one or more physical segments of memory. All segments in a physical segment are collectively called a "group."

All programs have at least two groups: a text group and a data group. Each group has a name. The text group is named "IGROUP" and the data group is named "DGROUP". The C compiler automatically applies these names to the text and data segments in each module. Thus, when the modules are eventually linked, all text segments belong to the same group, and all data segments belong to the same group.

Since a group corresponds to one physical segment, programs having more than 64 Kbytes each of text or data must be directed to two or more groups. (The limit per physical segment is 64 Kbytes.) You can create new groups by using the **-NGT** and **-NGD** options. The options have the form

-NGT *name*

and

-NGD *name*

where *name* is any combination of letters and digits. These options set the group name for the text and data segments of the given modules. All segments with the same name are loaded in the same physical segment. There will be one physical segment for each group named.

Compiler Summary

The following sections summarize **cc** options and memory models.

cc Options

The following is a complete list of **cc** options:

-c Creates a linkable object file for each source file.

-C Preserves comments when preprocessing a file (only when **-P** or -E).

-D *name* [=*string*]
Defines *name* to the preprocessor. The value is *string* or 1.

-E Preprocesses each source file, copying the result to the standard output.

-F *num*
Sets the size of the program stack to *num* bytes. (Default is 4,096 bytes.)

-i Creates separate instruction and data spaces for small model programs.

-I *pathname*
Adds *pathname* to the list of directories to be searched for **#include** files.

-K Removes stack probes from a program.

-l *name*
Searches library *name* for unresolved function names.

-L Creates an assembler listing file containing assembled code and assembly source instructions.

-M *string*
Sets the program configuration. *string* may be any combination of "s" (small model), "m" (middle model), "l" (large model), "e" (enable far and near keywords), "2" (enables 286 code generation), "b" (reverse word order), and "t" (sets data threshold for largest item in a segment). The "s", "m", and "l" choices are mutually exclusive.

-ND *name*
Sets the data segment name.

-NGD *name*
Sets the data group name.

-NGT *name*
Sets the text group name.

-nl *num*
Sets the maximum number of significant characters in external names. The default is 8; external names should be distinct in the first eight characters.

-NM *name*
Sets the module name.

-NT *name*
Sets the text segment name.

-o *filename*
Makes *filename* the name of the final executable program.

-O Invokes the object code optimizer.

-p Adds code for program profiling.

-P Preprocesses source files and sends output to files with the extension ".i".

-S Creates an assembly source listing.

-V *string*
Copies *string* to the object file.

-w Suppresses compiler warning messages.

-W *num*
Sets the output level for compiler warning messages.

-X Removes the standard directories from the list of directories to be searched for **#include** files.

Memory Models

Table 2-2 defines the number of text and data segments for the different program memory models. This table also lists the segment register values.

Table 2-2. Segments in Program Memory Models

Model	Text	Data	Segment Registers
Small	1*	1*	CS = DS = SS
Middle	1 per module	1	DS = SS
Large	1 per module	1 per module	

*In impure-text small model programs, text and data occupy the same segment. In pure-text programs, they occupy different segments.

Pointer and Integer Sizes

Table 2-3 defines the sizes (in bits) of integers (**int** type), and text and data pointers, in each program memory model.

Table 2-3. Pointer and Integer Sizes in Program Memory Models

Model	Data Pointer	Text Pointer	Integer
Small	16	16	16
Middle	16	32	16
Large	32	32	16

Segment and Module Names

Table 2-4 lists the default text and data segment names, and the default module name, for each object file.

Table 2-4. Default Segment and Module Names

Model	Text	Data	Module
Small	_TEXT	_DATA	*filename*
Middle	*module*_TEXT	_DATA	*filename*
Large	*module*_TEXT	*module*_DATA	*filename*

intel®

CHAPTER 3
lint: C PROGRAM CHECKER

This chapter explains how to use the C program checker **lint.** The program examines C source files and warns of errors or misconstructions that may cause errors during compilation of the file or during execution of the compiled file.

In particular, **lint** checks for

- Unused functions and variables
- Unknown values in local variables
- Unreachable statements and infinite loops
- Unused and misused return values
- Inconsistent types and type casts
- Mismatched types in assignments
- Nonportable and old-fashioned syntax
- Strange constructions
- Inconsistent pointer alignment and expression evaluation order

The **lint** program and the C compiler are generally used together to check and compile C language programs. Although the C compiler rapidly and efficiently compiles C language source files, it does not perform the sophisticated type and error checking required by many programs. The **lint** program, on the other hand, provides thorough checking of source files without compiling.

Invoking lint

You can invoke **lint** by typing its name at the shell command prompt. The command has the form

lint [*option*] ... *filename* ... *lib* ...

where *option* is a command option that defines how the checker should operate, *filename* is the name of a C language source file to be checked, and *lib* is the name of a library to check. You can give more than one option, file name, or library name in the command as long as you use spaces to separate them. If you give two or more file names, **lint** assumes that the files form a complete program and checks the files accordingly. For example, the command

```
lint main.c add.c
```

treats **main.c** and **add.c** as two parts of a complete program.

If **lint** discovers errors or inconsistencies in a source file, it produces messages describing the problem. The message has the form

filename (*num*): *description*

where *filename* is the name of the source file containing the problem, *num* is the number of the line in the source containing the problem, and *description* is a description of the problem. For example, the message

```
main.c (3): warning: x unused in function main
```

shows that the variable **x**, defined in line 3 of the source file **main.c**, is not used anywhere in the file.

Checking for Unused Variables and Functions

The **lint** program checks for unused variables and functions by seeing if each declared variable and function is used at least once in the source file. The program considers a variable or function used if the name appears in at least one statement. It is not considered used if it only appears on the left side of an assignment. For example, in the following program fragment

```
main ()
{
        int  x, y, z;

        x = 1;  y = 2;  z = x + y;
```

the variables **x** and **y** are considered used, but variable **z** is not.

Unused variables and functions often occur during the development of large programs. It is not uncommon for a programmer to remove all references to a variable or function from a source file but forget to remove its declaration. Such unused variables and functions rarely cause working programs to fail but do make programs harder to understand and change. Checking for unused variables and functions can also help you find variables or functions that you intended to used but have accidentally left out of the program.

Note that the **lint** program does not report a variable or function unused if it is explicitly declared with the **extern** storage class. Such a variable or function is assumed to be used in another source file.

You can direct **lint** to ignore all the external declarations in a source file by using the **-x** (for "external") option. The option causes the program checker to skip any line that begins with the **extern** storage class.

The option is typically used to save time when checking a program, especially if all external declarations are known to be valid.

Some programming styles require functions that perform closely related tasks to have the same number and type of arguments regardless of whether or not these arguments are used. Under normal operation, **lint** reports any argument not used as an unused variable, but you can direct **lint** to ignore unused arguments by using the **-v** option. The **-v** option causes **lint** to ignore all unused function arguments except for those declared with **register** storage class. The program considers unused arguments of this class to be a preventable waste of the register resources of the computer.

You can direct **lint** to ignore all unused variables and functions by using the **-u** (for "unused") option. This option prevents **lint** from reporting variables and functions it considers unused.

This option is typically used when checking a source file that contains just a portion of a large program. Such source files usually contain declarations of variables and functions to be used in other source files that are not explicitly used within the file. Since **lint** can check only the given file, it normally assumes that such variables or functions are unused and reports them as errors.

Checking Local Variables

The **lint** program checks all local variables to see that they are set to a value before being used. Since local variables have either automatic or register storage class, their values at the start of the program or function cannot be known. Using such a variable before assigning a value to it is an error.

The program checks the local variables by searching for the first assignment in which the variable receives a value and the first statement or expression in which the variable is used. If the first assignment appears later than the first use, **lint** considers the variable inappropriately used. For example, in the program fragment

```
int c;

if ( c != EOT )
        c = getchar();
```

lint warns that the the variable **c** is used before it is assigned.

If the variable is used in the same statement in which it is assigned for the first time, **lint** determines the order of evaluation of the statement and displays an appropriate message. For example, in the program fragment

```
int i, total;

scanf("%d", &i);
total = total + i;
```

lint warns that the variable "total" is used before it is set since it appears on the right side of the same statement that assigns its first value.

Static and external variables are always initialized to zero before program execution begins, so **lint** does not report such variables if they are used before being set to a value.

Checking for Unreachable Statements

The **lint** program checks for unreachable statements, that is, for unlabeled statements that immediately follow a **goto, break, continue,** or **return** statement. During execution of a program, the unreachable statements never receive execution control and are therefore considered wasteful. For example, in the program fragment

```
int x,y;

return (x + y);
exit (1);
```

the function call **exit** after the **return** statement is unreachable.

Unreachable statements are common when developing programs containing large **switch** statements or loops containing **break** and **continue** statements. Such statements are wasteful and should be removed.

During normal operation, **lint** reports all unreachable **break** statements. Unreachable **break** statements are relatively common (some programs created by the **yacc** and **lex** programs contain hundreds), so it may be desirable to suppress these reports. You can direct **lint** to suppress the reports by using the **-b** option.

Note that **lint** assumes that all functions eventually return control, so it does not report as unreachable any statement that follows a function that takes control and never returns it. For example:

```
exit (1);
return;
```

The call to **exit** causes the **return** statement to become an unreachable statement, but **lint** does not report it as such.

Checking for Infinite Loops

The **lint** program checks for infinite loops and for loops that are never executed. For example, the statements

```
while (1) { }
```

and

```
for (;;) { }
```

are both considered infinite loops. While the statements

```
while (0) { }
```

and

```
for (;0;) { }
```

are never executed.

Although some valid programs have such loops, they are generally considered errors.

Checking Function Return Values

The **lint** program checks that a function returns a meaningful value if necessary. Some functions return values that are never used; some programs incorrectly use function values that have never been returned. **lint** addresses these problems in a number of ways.

Within a function definition, the appearance of both

```
return (expr);
```

and

```
return;
```

statements is cause for alarm. In this case, **lint** produces the following error message:

```
function name contains return(e) and return
```

It is difficult to detect when a function return is implied by the flow of control reaching the end of the given function. This is demonstrated with a simple example:

```
f(a)
{
        if (a)
                return (3);
        g();
}
```

Note that if the variable **a** tests false, **f** will call the function **g** and then return with no defined return value. This will trigger a report from **lint.** If **g,** like **exit,** never returns, the message will still be produced when in fact nothing is wrong. In practice, potentially serious bugs can be discovered with this feature. It also accounts for a substantial fraction of the noise messages produced by **lint.**

Checking for Unused Return Values

The **lint** program checks for cases where a function returns a value, but the value is rarely if ever used. **lint** considers functions that return unused values to be inefficient and functions that return rarely used values to be a result of bad programming style.

lint also checks for cases where a function does not return a value but the value is used anyway. This is considered a serious error.

Checking Types

lint enforces the type checking rules of C more strictly than the C compiler. The additional checking occurs in four major areas:

- Across certain binary operators and implied assignments
- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations

A number of operators, including the assignment, conditional, and relational operators, have an implied balancing between types of operands. The argument of a **return** statement and expressions used in initialization also suffer similar conversions. In these operations, **char, short, int, long, unsigned, float,** and **double** types can be freely intermixed. The types of pointers must agree exactly, except that arrays of x's can be intermixed with pointers to x's.

The type checking rules also require that, in structure references, the left operand of a pointer arrow symbol (->) be a pointer to a structure, the left operand of a period (.) be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char, short, int,** and **unsigned.** Pointers can also be matched with the associated arrays. Aside from these relaxations in type checking, all actual arguments must agree in type with their declared counterparts.

For enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations, and that the only operations applied are assignment (=), initialization, equals (==), and not-equals (!=). Enumerations may also be function arguments and return values.

Checking Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** reports this as suspect. But consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. On the other hand, if this code is moved to another machine, it should be looked at carefully. The **-c** option controls the printing of comments about casts. When **-c** is in effect, casts are not checked and all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Checking for Nonportable Character Use

lint flags certain comparisons and assignments as illegal or nonportable. For example, the fragment

```
char c;
.
.
.
if( (c = getchar( ) ) < 0 ) ...
```

works on some machines but fails on machines where characters always take on positive values. The solution is to declare **c** an integer, since **getchar** is actually returning integer values. In any case, **lint** issues the message

```
nonportable character comparison
```

A similar issue arises with bitfields. When assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true where on some machines bitfields are considered as signed quantities. While it may seem counter-intuitive to consider that a 2-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned.**

Checking for Assignment of longs to ints

Bugs may arise from the assignment of a **long** to an **int,** because of a loss in accuracy in the process. This may happen in programs that have been incompletely converted by changing type definitions with **typedef.** When a **typedef** variable is changed from **int** to **long,** the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, you may wish to suppress detection of these assignments by using the **-a** option.

Checking for Strange Constructions

Several perfectly legal but somewhat strange constructions are flagged by **lint.** The generated messages encourage better code quality and clearer style, and may even point out bugs. For example, in the statement

```
*p + + ;
```

the asterisk (*) does nothing and **lint** prints

```
null effect
```

The program fragment

```
unsigned x;
if (x < 0) ...
```

is also strange since the test will never succeed. Similarly, the test

```
unsigned x;
if (x > 0) ...
```

is equivalent to

```
unsigned x;
if( x != 0) ...
```

which may not be the intended action. In these cases, **lint** prints the message

```
degenerate unsigned comparison
```

If you use

```
if( 1 != 0 ) ...
```

then **lint** reports

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs that arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

and

```
x<<2 + 40
```

probably do not do what is intended. The best solution is to parenthesize such expressions. **lint** encourages this by printing an appropriate message.

Finally, **lint** checks variables redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered bad style, usually unnecessary, and frequently a bug.

If you do not wish these heuristic checks, you can suppress them by using the **-h** option.

Checking for Use of Older C Syntax

lint checks for older C constructions. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) can cause ambiguous expressions, such as

```
a=-1;
```

which could be taken as either

```
a =- 1;
```

or

```
a = -1;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, -=) have no such ambiguities. To encourage the abandonment of the older forms, **lint** checks for occurrences of these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize **x** to 1. This causes syntactic difficulties. For example

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration

```
int x ( y ) { . . .
```

and the compiler must read past "x" to determine what the declaration really is. The problem is even more perplexing when the initializer involves a macro. The current C syntax places an equal sign between the variable and the initializer:

```
int x = -1 ;
```

This form is free of any possible syntactic ambiguity.

Checking Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others, due to alignment restrictions. For example, on some machines it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On other machines, however, double precision values must begin on even word boundaries; thus, not all such assignments make sense. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

```
possible pointer alignment problem
```

results from this situation.

Checking Expression Evaluation Order

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines in which the stack runs backward, function arguments will probably be best evaluated from right to left; on machines with a stack running forward, left to right is probably best. Function calls embedded as arguments of other functions may or may not be treated in the same way as ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

So that the efficiency of C on a particular machine is not unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the compiler, and various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i + + ] ;
```

will draw the comment

```
warning: i evaluation order undefined
```

Embedding Directives

There are occasions when the programmer is smarter than **lint.** There may be valid reasons for illegal type casts, functions with a variable number of arguments, and other constructions that **lint** finds objectionable. Moreover, as specified in the above sections, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Some way of communicating with **lint,** typically to turn off its output, is desirable. Therefore, a number of words are recognized by **lint** when they are embedded in comments in a C source file. These words are called directives. **lint** directives are invisible to the compiler.

The first directive discussed concerns flow of control information. If a particular place in the program cannot be reached, this can be asserted at the appropriate spot in the program with the directive

```
/* NOTREACHED */
```

Similarly, if you desire to turn off strict type checking for the next expression, use the directive

```
/* NOSTRICT */
```

The situation reverts to the previous default after the next expression. The **-v** option can be turned on for one function with the directive

```
/* ARGSUSED */
```

Comments about a variable number of arguments in calls to a function can be turned off by preceding the function definition with the directive

```
/* VARARGS */
```

In some cases, you may want to check the first several arguments and leave the later arguments unchecked. Do this by following the VARARGS keyword immediately with a digit giving the number of arguments that should be checked. Thus

```
/* VARARGS2 */
```

causes only the first two arguments to be checked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file, discussed in the next section.

Checking for Library Compatibility

lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This testing is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The "VARARGS" and "ARGSUSED" directives can be used to specify features of the library functions.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions defined in a library file but not used in a source file, draw no comments. **lint** does not simulate a full library search algorithm. **lint** does check to see if the source files contain redefinitions of library routines.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the standard functions that are normally loaded when a C program is run. When the **-p** option is in effect, the portable libarary file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

intel®

CHAPTER 4
make: PROGRAM MAINTAINER

The **make** program provides an easy way to automate the creation of large programs. **make** reads commands from a user-defined "makefile" that lists the files to be created, the commands that create them, and the files from which they are created. When you direct **make** to create a program, it makes sure that each file on which the program depends is up to date, then creates the program by executing the given commands. If a file is not up to date, **make** updates it before creating the program by executing explicitly given commands or one of the many built-in commands.

This chapter explains how to use **make** to automate medium-sized programming projects. It explains how to create makefiles for each project and how to invoke **make** for creating programs and updating files. For more details about the program, see **make** in Appendix B, "Programming Commands."

Creating a Makefile

A makefile contains one or more lines of text called dependency lines. A dependency line shows how a given file depends on other files and what commands are required to bring a file up to date. A dependency line has the form

target ... : [*dependent* .. .] [; *command* ...]

where *target* is the name of a file to be updated, *dependent* is the name of a file on which the target depends, and *command* is the XENIX command needed to create the target file. Each dependency line must have at least one command associated with it, even if it is only the null command (;).

You may give more than one target name or dependent name if desired. Each name must be separated from the next by at least one space. The target names must be separated from the dependent names by a colon (:). File names must be spelled as defined by the XENIX system. Shell metacharacters, such as star (*) and question mark (?), in file names are expanded when **make** uses the name.

You may give a sequence of commands on the same line as the target and dependent names if you precede each command with a semicolon (;). You can give additional commands on following lines by beginning each line with a tab character. Commands must be given exactly as they would appear on a shell command line. The at sign (@) may be placed in front of a command to prevent **make** from displaying the command before executing it. Shell commands, such as **cd**, must appear on single lines; they must not contain the backslash (\) and newline character combination that normally can be used to continue commands between lines.

You may add a comment to a makefile by starting the comment with a number sign (#) and ending it with a newline character. All characters after the number sign are ignored. Comments may be placed at the end of a dependency line if desired. If a command contains a number sign, it must be enclosed in double quotation marks ("#").

If a dependency line is too long, you can continue it by typing a backslash (\) immediately followed by a newline character.

The makefile should be kept in the same directory as the given source files. For convenience, the file names **makefile, Makefile, s.makefile,** and **s.Makefile** are provided as default file names used by **make** if no explicit name is given at invocation. You may use one of these names for your makefile or choose one of your own. If the file name begins with the **s.** prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the latest version of the file.

To illustrate dependency lines, consider the following example. A program named **prog** is made by linking three object files, **x.o, y.o,** and **z.o.** These object files are created by compiling the C language source files **x.c, y.c,** and **z.c.** Furthermore, the files **x.c** and **y.c** contain the line

```
#include "defs"
```

This means that **prog** depends on the three object files, the object files depend on the C source files, and two of the source files depend on the include file **defs.** You can represent these relationships in a makefile with the following lines

```
prog: x.o y.o z.o
	cc x.o y.o z.o -o prog

x.o: x.c defs
	cc -c x.c

y.o: y.c defs
	cc -c y.c

z.o: z.c
	cc -c z.c
```

In the first dependency line, **prog** is the target file and **x.o, y.o,** and **z.o** are its dependents. The command sequence

```
cc x.o y.o z.o -o prog
```

on the next line tells how to create **prog** if it is out of date. The program is out of date if any one of its dependents has been modified since **prog** was last created.

The second, third, and fourth dependency lines have the same form, with the **x.o, y.o,** and **z.o** files as targets and **x.c, y.c, z.c,** and **defs** files as dependents. Each dependency line has one command sequence that defines how to update the given target file.

Invoking make

Once you have a makefile and wish to update and modify one or more target files in the directory, you can invoke **make** by typing its name and optional arguments. The invocation has the form

make [*option*] ... [*macdef*] ... [*target*] ...

where *option* is a program option used to modify program operation, *macdef* is a macro definition used to give a macro a value or meaning, and *target* is the file name of a file to be updated. *target* must correspond to one of the target names in the makefile. All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct **make** to update the first target file in the makefile by typing just the program name "make". In this case, **make** searches for one of the files **makefile, Makefile, s.makefile,** or **s.Makefile** in the current directory and uses the first one it finds as the makefile. For example, assume that the current makefile contains the dependency lines given in the last section. Then the command

make

compares the modification dates of the **prog** program and each of the object files **x.o, y.o,** and **z.o** and recreates **prog** if any changes have been made to any object file since **prog** was last created. It also compares the modified dates of the object files with those of the four source files, **x.c, y.c, z.c,** and **defs,** and recreates the object files if the source files have changed. It does this before recreating **prog** so that the recreated object files can be used to recreate **prog.** If none of the source or object files has been altered since the last time **prog** was made, **make** announces this fact and stops. No files are changed.

You can direct **make** to update a given target file by giving the file name of the target. For example,

make x.o

causes **make** to recompile the **x.o** file if the **x.c** or **defs** files have changed since the object file was last created. Similarly, the command

make x.o z.o

causes **make** to recompile **x.o** and **z.o** if the corresponding dependents have been modified. **make** processes target names from the command line in a left-to-right order.

You can specify the name of the makefile you wish **make** to use by giving the **-f** option in the invocation. The option has the form

-f *filename*

where *filename* is the name of the makefile. You must supply a full path name if the file is not in the current directory. For example, the command

make -f makeprog

reads the dependency lines of the makefile named **makeprog** found in the current directory. You can direct **make** to read dependency lines from the standard input by giving "- " as the file name. **make** will read the standard input until end-of-file is encountered.

You may use program options to modify the operation of the **make** program. The following list describes some of the options:

-p Prints the complete set of macro definitions and dependency lines in a makefile.

-i Ignores errors returned by XENIX commands.

-k Abandons work on the current entry but continues on other branches that do not depend on that entry.

-s Executes commands without displaying them.

-r Ignores the built-in rules.

-n Displays commands but does not execute them. **make -n** even displays lines beginning with the at sign (@).

-e Ignores any macro definitions that attempt to assign new values to the shell's environment variables.

-t Changes the modification date of each target file without recreating the files.

Note that **make** executes each command in the makefile by passing it to a separate invocation of a shell. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed. If an error occurs, **make** normally stops the command.

Using Pseudo-Target Names

It is often useful to include dependency lines that have pseudo-target names, i.e., names for which no files actually exist or are produced. Pseudo-target names allow **make** to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name "cleanup" is given in the invocation of **make.**

```
cleanup :
        rm x.o y.o z.o
```

Since no file exists for a given pseudo-target name, the target is always assumed to be out of date. Thus the associated command is always executed.

make also has built-in pseudo-target names that modify its operation. The pseudo-target name ".IGNORE" causes **make** to ignore errors during execution of commands, allowing **make** to continue after an error. This is the same as the **-i** option. (**make** also ignores errors for a given command if the command string begins with a hyphen (-).)

The pseudo-target name ".DEFAULT" defines the commands to be executed when no built-in rule or user-defined dependency line exists for the given target. You may give any number of commands with this name. If ".DEFAULT" is not used and an undefined target is given, **make** prints a message and stops.

The pseudo-target name ".PRECIOUS" prevents dependents of the current target from being deleted when **make** is terminated using the INTERRUPT or QUIT key, and the pseudo-target name ".SILENT" has the same effect as the **-s** option.

Using Macros

An important feature of a makefile is that it can contain macros. A macro is a short name that represents a file name or command option. The macros can be defined when you invoke **make** or in the makefile itself.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll  -ly
LIBES =
```

The last definition assigns "LIBES" the null string. A macro that is never explicitly defined has the null string as its value.

A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be placed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

Macros are typically used as placeholders for values that may change from time to time. For example, the following makefile uses two macros for the names of object files to be linked and for the name of the library.

```
OBJECTS = x.o  y.o  z.o
LIBES = -lln
prog:  $(OBJECTS)
       cc  $(OBJECTS)  $(LIBES)  -o prog
```

If this makefile is invoked with the command

```
make
```

it will link the three object files with the **lex** library specified with the **-lln** switch.

You may include a macro definition in a command line. A macro definition argument has the same form as a macro definition in a makefile. Macros in a command line override corresponding definitions found in the makefile. For example, the command

```
make "LIBES = -lln -lm"
```

assigns the library options **-lln** and **-lm** to **LIBES.** (It is necessary to quote arguments with embedded blanks in XENIX commands.)

You can modify all or part of the value generated from a macro invocation without changing the macro itself by using the substitution sequence. The sequence has the form

name : *st1* = [*st2*]

where *name* is the name of the macro whose value is to be modified, *st1* is the character or characters to be modified, and *st2* is the character or characters to replace the modified characters. If *st2* is not given, *st1* is replaced by a null character.

The substitution sequence is typically used to allow user-defined metacharacters in a makefile. For example, suppose that ".x" is to be used as a metacharacter for a prefix and suppose that a makefile contains the definition

```
FILES = prog1.x  prog2.x  prog3.x
```

Then the macro invocation

```
$(FILES  :  .x = .o)
```

generates the value

```
prog1.o  prog2.o  prog3.o
```

The actual value of **FILES** remains unchanged.

make has five built-in macros that can be used when writing dependency lines. The following is a list of these macros:

$* Contains the name of the current target with the suffix removed. Thus if the current target is **prog.o**, **$*** contains **prog**. It may be used in dependency lines that redefine the built-in rules.

$@ Contains the full path name of the current target. It may be used in dependency lines with user-defined target names.

$< Contains the file name of the dependent that is more recent than the given target. It may be used in dependency lines with built-in target names or the **.DEFAULT** pseudo-target name.

$? Contains the file names of the dependents that are more recent than the given target. It may be used in dependency lines with user-defined target names.

$% Contains the file name of a library member. It may be used with target library names (see the section "Using Libraries" later in this chapter). In this case, **$@** contains the name of the library and **$%** contains the name of the library member.

You can change the meaning of a built-in macro by appending the **D** or **F** descriptor to its name. A built-in macro with the **D** descriptor contains the name of the directory containing the given file. If the file is in the current directory, the macro contains ".". A macro with the **F** descriptor contains the name of the given file with the directory name part removed. The **D** and **F** descriptors must not be used with the **$?** macro.

Using Shell Environment Variables

make provides access to current values of the shell's environment variables such as **HOME, PATH,** and **LOGIN. make** automatically assigns the value of each shell variable in your environment to a macro of the same name. You can access a variable's value in the same way that you access the value of explicitly defined macros. For example, in the following dependency line, "$(HOME)" has the same value as the user's **HOME** variable.

```
prog :
        cc  $(HOME)/x.o  $(HOME)/y.o /usr/pub/z.o
```

make assigns the shell variable values after it assigns values to the built-in macros, but before it assigns values to user-specified macros. Thus, you can override the value of a shell variable by explicitly assigning a value to the corresponding macro. For example, the following macro definition causes **make** to ignore the current value of the **HOME** variable and use **/usr/pub** instead.

```
HOME = /usr/pub
```

If a makefile contains macro definitions that override the current values of the shell variables, you can direct **make** to ignore these definitions by using the **-e** option.

make has use of two shell variables, **MAKE** and **MAKEFLAGS,** that correspond to two special-purpose macros.

The **MAKE** macro provides a way to override the **-n** option and execute selected commands in a makefile. When **MAKE** is used in a command, **make** will always execute that command, even if **-n** has been given in the invocation. The variable may be set to any value or command sequence.

The **MAKEFLAGS** macro contains one or more **make** options and can be used in invocations of **make** from within a makefile. You may assign any **make** options to **MAKEFLAGS** except **-f**, **-p**, and **-d**. If you do not assign a value to the macro, **make** automatically assigns the current options to it, i.e., the options given in the current invocation.

The **MAKE** and **MAKEFLAGS** macros, together with the **-n** option, are typically used to debug makefiles that generate entire software systems. For example, in the following makefile, setting **MAKE** to "make" and invoking this file with the **-n** option displays all the commands used to generate the programs **prog1**, **prog2**, and **prog3** without actually executing them.

```
system : prog1 prog2 prog3
        @echo System complete.

prog1 : prog1.c
        $(MAKE) $(MAKEFLAGS) prog1

prog2 : prog2.c
        $(MAKE) $(MAKEFLAGS) prog2

prog3 : prog3.c
        $(MAKE) $(MAKEFLAGS) prog3
```

Using the Built-In Rules

make provides a set of built-in dependency lines, called built-in rules, that automatically check the targets and dependents given in a makefile and create up-to-date versions of these files if necessary. The built-in rules are identical to user-defined dependency lines except that they use the suffix of the file name as the target or dependent instead of the file name itself. For example, **make** automatically assumes that all files with the suffix **.o** have dependent files with the suffixes **.c** and **.s**.

When no explicit dependency line is given in a makefile for a given file, **make** automatically checks the default dependents of the file, forming the name of the dependents by removing the suffix of the given file and appending the predefined dependent suffixes. If the given file is out of date with respect to these default dependents, **make** searches for a built-in rule that defines how to create an up to date version of the file and executes it. There are built-in rules for the following files:

.o Object file
.c C source file
.r Ratfor source file
.f Fortran source file

.s Assembler source file
.y Yacc-C source grammar
.yr Yacc-Ratfor source grammar
.l Lex source grammar

For example, if the file **x.o** is needed and there is an **x.c** in the description or directory, **x.c** is compiled. If there is also an **x.l**, that grammar would be run through **lex** before compiling the result.

The built-in rules are designed to reduce the size of your makefiles. They provide the rules for creating common files from typical dependents. Reconsider the example given in the section "Creating a Makefile." In this example, the program **prog** depended on three object files, **x.o, y.o,** and **z.o.** These files in turn depended on the C language source files **x.c, y.c,** and **z.c.** The files **x.c** and **y.c** also depended on the include file **defs.** In the original example each dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files:

```
prog: x.o y.o z.o
      cc x.o y.o z.o -o prog

x.o y.o: defs
```

In this makefile, **prog** depends on three object files, and an explicit command is given showing how to update **prog.** However, the second line merely shows that two object files depend on the include file **defs.** No explicit command sequence is given on how to update these files if necessary. Instead, **make** uses the built-in rules to locate the desired C source files, compile these files, and create the necessary object files.

Changing the Built-In Rules

You can change the built-in rules by redefining the macros used in these lines or by redefining the commands associated with the rules. You can display a complete list of the built-in rules and the macros used in the rules by typing

```
make -fp - 2>/dev/null </dev/null
```

The rules and macros are displayed at the standard output.

The macros of the built-in dependency lines define the names and options of the compilers, program generators, and other programs invoked by the built-in commands. **make** automatically assigns a default value to these macros when you start the program. You can change the values by redefining the macro in your makefile. For example, the following built-in rule contains three macros, **CC, CFLAGS,** and **LOADLIBES.**

```
.c :
        $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
```

You can redefine any of these macros by placing the appropriate macro definition at the beginning of the makefile.

You can redefine the action of a built-in rule by giving a new rule in your makefile. A built-in rule has the form

```
suffix-rule :
        command
```

where *suffix-rule* is a combination of suffixes showing the relationship of the implied target and dependent, and *command* is the XENIX command required to carry out the rule. If more than one command is needed, they are given on separate lines.

The new rule must begin with an appropriate *suffix-rule.* The available *suffix-rules* are

.c	.c
.sh	.sh
.c.o	.c.o
.c.c	.s.o
.s.o	.y.o
.y.o	.l.o
.l.o	.y.c
.y.c	.l.c
.c.a	.c.a
.s.a	.h.h

A tilde (~) indicates an SCCS file. A single suffix indicates a rule that makes an executable file from the given file. For example, the suffix rule ".c" is for the built-in rule that creates an executable file from a C source file. A pair of suffixes indicates a rule that makes one file from the other. For example, ".c.o" is for the rule that creates an object file (**.o**) from a corresponding C source file (**.c**).

Any commands in the rule may use the built-in macros provided by **make.** For example, the following dependency line redefines the action of the **.c.o** rule.

```
.c.o :
  cc86 $< -c $*.o
```

If necessary, you can also create new *suffix-rules* by adding a list of new suffixes to a makefile with **.SUFFIXES.** This pseudo-target name defines the suffixes that may be used to make *suffix-rules* for the built-in rules. The line has the form

```
.SUFFIXES:  suffix ...
```

where *suffix* is usually a lowercase letter preceded by a dot (.). If more than one suffix is given, you must use spaces to separate them.

The order of the suffixes is significant. Each suffix is a dependent of the suffixes preceding it. For example, the suffix list

```
.SUFFIXES:  .o  .c  .y  .l  .s
```

causes **prog.c** to be a dependent of **prog.o,** and **prog.y** to be a dependent of **prog.c,** etc.

You can create new *suffix-rules* by combining dependent suffixes with the suffix of the intended target. The dependent suffix must appear first.

If a **SUFFIXES** list appears more than once in a makefile, the suffixes are combined into a single list. If a **SUFFIXES** is given that has no list, all suffixes are ignored.

Using Libraries

You can direct **make** to use a file contained in an archive library as a target or dependent. To do this you must explicitly name the file you wish to access by using a library name. A library name has the form

```
lib(member-name)
```

where *lib* is the name of the library containing the file, and *member-name* is the name of the file. For example, the library name

```
libtemp.a(print.o)
```

refers to the object file **print.o** in the archive library **libtemp.a.**

You can create your own built-in rules for archive libraries by adding the **.a** suffix to the suffix list and creating new suffix combinations. For example, the combination ".c.a" may be used for a rule that defines how to create a library member from a C source file. Note that the dependent suffix in the new combination must be different than the suffix of the ultimate file. For example, the combination ".c.a" can be used for a rule that creates **.o** files, but not for one that creates **.c** files.

The most common use of the library naming convention is to create a makefile that automatically maintains an archive library. For example, the following dependency lines define the commands required to create a library, named **lib,** containing up to date versions of the files **file1.o, file2.o,** and **file3.o.**

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date

.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@$*.o
        rm -f $*.o
```

The **.c.a** rule shows how to redefine a built-in rule for a library. In the following example, the built-in rule is disabled, allowing the first dependency to create the library.

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?
        @echo lib is now up to date
.c.a:;
```

In this example, a substitution sequence is used to change the value of the **$?** macro from the names of the object files "file1.o", "file2.o", and "file3.o" to "file1.c", "file2.c", and "file3.c".

Troubleshooting

Most difficulties in using **make** arise from **make**'s specific meaning of dependency. If the file **x.c** has the line

```
#include "defs"
```

then the object file **x.o** depends on **defs;** the source file **x.c** does not. (If **defs** is changed, it is not necessary to do anything to the file **x.c,** while it is necessary to recreate **x.o.**)

To determine which commands **make** will execute, without actually executing them, use the **-n** option. For example, the command

```
make -n
```

prints out the commands **make** would normally execute without actually executing them.

If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the **-t** (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make -ts
```

which stands for touch silently, causes the relevant files to appear up to date.

The debugging option **-d** causes **make** to print out a very detailed description of what it is doing, including the file times. The output is verbose and is recommended only as a last resort.

Using make: An Example

As an example of the use of **make,** examine the makefile used to maintain the **make** program itself. The code for **make** is spread over a number of C source files and a **yacc** grammar.

```
# Description file for the make command

# Macro definitions below
P = lpr
FILES = Makefile vers.c defs main.c doname.c misc.c files.c dosys.c\
        gram.y lex.c
OBJECTS = vers.o main.o ... dosys.o  gram.o
LIBES =
LINT = lint -p
CFLAGS = -O

#targets: dependents
##ITAB>actions

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make
```

```
print:  $(FILES)     # print recently changed files
        pr  $? | $P
        touch  print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c vers.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c vers.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the **make** source and makefile:

```
cc      -c vers.c
cc      -c main.c
cc      -c doname.c
cc      -c misc.c
cc      -c files.c
cc      -c dosys.c
yacc    gram.y
mv      y.tab.c gram.c
cc      -c gram.c
cc      vers.o main.o ... dosys.o gram.o -o make
13188 + 3348 + 3044 = 19580b = 046174b
```

Although none of the source files or grammars was mentioned by name in the makefile, **make** found them by using its suffix rules and issued the needed commands. The string of digits results from the **size make** command.

The last few targets in the makefile are useful maintenance sequences. The **print** target prints only the files that have been changed since the last **make print** command. A zero-length file **print** is maintained to keep track of the time of the printing; the **$?** macro in the command line then picks up only the names of the files changed since **print** was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro.

intel®

CHAPTER 5
SCCS: SOURCE CODE CONTROL SYSTEM

The Source Code Control System (SCCS) is a collection of XENIX commands that create, maintain, and control special files called SCCS files. The SCCS commands can create and store multiple versions of a program or document in a single file instead of in one file for each version. The commands can retrieve any version you wish at any time, make changes to this version, and save the changes as a new version of the file in the SCCS file.

The SCCS system is useful wherever you require a compact way to store multiple versions of the same file. SCCS provides an easy way to update any given version of a file and explicitly record the changes made. The commands are typically used to control changes to multiple versions of source programs, but may also be used to control multiple versions of manuals, specifications, or other documentation.

This chapter explains how to make SCCS files, how to update the files contained in SCCS files, and how to maintain SCCS files once they are created.

Basic Information

This section provides some basic information about SCCS. In particular, it describes

- SCCS files and directories
- Deltas and SIDs
- SCCS working files
- SCCS command arguments
- File administration

Files and Directories

All SCCS files (also called s-files) are originally created from text files containing documents or programs created by a user. The text files must have been created using a XENIX text editor such as **vi**. Special characters in the files are allowed only if they are also allowed by the given editor.

To simplify s-file storage, all logically related files (e.g., files belonging to the same project) should be kept in the same directory. Such directories should contain only s-files and should have read and search permission for everyone and write permission for the user alone.

Note that you must not use the XENIX **link** command to create multiple links to an s-file.

Deltas and SIDs

Unlike an ordinary text file, an s-file contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. The lists are combined to create the desired version from the original.

Each list of changes is called a delta. Each delta has an identification string called an SID. The SID is a string of at least two and at most four numbers separated by periods. The numbers name the version and define how it is related to other versions. For example, the first delta is usually numbered "1.1" and the second "1.2".

The first number in any SID is called the release number. The release number usually indicates a group of versions that are similar and generally compatible. The second number in the SID is the level number. It indicates major differences between files in the same release.

An SID may also have two optional numbers. The branch number, the optional third number, indicates changes at a particular level, and the sequence number, the fourth number, indicates changes at a particular branch. For example, the SIDs "1.1.1.1" and "1.1.1.2" indicate two new versions that contain slight changes to the original delta 1.1.

An s-file may at any time contain several different releases, levels, branches, and sequences of the same file. In general, the maximum number of releases an s-file may contain is 9999, that is, release numbers may range from 1 to 9999. The same limit applies to level, branch, and sequence numbers.

When you create a new version, SCCS usually creates a new SID by incrementing the level number of the original version. If you wish to create a new release, you must explicitly instruct the system to do so. A change to a release number indicates a major new version of the file. How to create a new version of a file and change release numbers is described later.

SCCS creates a branch and sequence number for the SID of a new version if the next higher level number already exists. For example, if you change version 1.3 to create a version 1.4 and then change 1.3 again, the SCCS system creates a new version named 1.3.1.1.

Version numbers can become quite complicated. In general, it is wise to keep the numbers as simple as possible by carefully planning the creation of each new version.

SCCS Working Files

SCCS uses several different kinds of files to complete its tasks. In general, these files contain information about the commands in progress or contain actual text. For convenience, SCCS names these files by placing a prefix before the name of the original file from which all versions are made. The following is a list of the working files:

s-file — Contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. The name of an s-file is formed by placing the prefix **s.** at the beginning of the original file name.

x-file — A temporary copy of the s-file. It is created by SCCS commands that change the s-file and is used instead of the s-file to carry out the changes. When all changes are complete, SCCS removes the original file and gives the x-file the name of the original s-file. The name of the x-file is formed by placing the prefix **x.** at the beginning of the original file name.

g-file — An ordinary text file created by applying the deltas in a given s-file to the original file. The g-file represents a copy of the given version of the original file and as such receives the same file name as the original. When created, a g-file is placed in the current working directory of the user who requested the file.

p-file — A special file containing information about the versions of an s-file currently being edited. The p-file is created when a g-file is retrieved from the s-file. The file remains until all currently retrieved files have been saved in the s-file. The p-file contains one or more entries describing the SID of the retrieved g-file, the proposed SID of the new, edited g-file, and the login name of the user who retrieved the g-file. The p-file name is formed by placing the prefix **p.** at the beginning of the original file name.

z-file — A lock file used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifies an SCCS file, it creates a z-file and copies its own process ID to it. Any other command that attempts to access the file while the z-file is present displays an error message and stops. When the original command has finished its tasks, it deletes the z-file before stopping. The z-file name is formed by placing the prefix **z.** at the beginning of the original file name.

l-file — A special file containing a list of the deltas required to create a given version of a file. The l-file name is formed by placing the prefix **l.** at the beginning of the original file name.

d-file — A temporary copy of the g-file used to generate a new delta.

q-file — A temporary file used by the **delta** command when updating the corresponding p-file. The q-file is not directly accessible.

In general, a user never directly accesses x-files, z-files, d-files, or q-files. If a system crash or similar situation abnormally terminates a command, the user may wish to delete these files before using the SCCS commands.

SCCS Command Arguments

Almost all SCCS commands accept two types of arguments: options and file names. These appear in the SCCS command line immediately after the command name.

An option indicates a special action to be taken by the given SCCS command. An option is usually a lowercase letter preceded by a minus sign (-). Some options require an additional name or value.

A file name indicates the file to be acted on. The syntax for SCCS file names is like other XENIX file names. Appropriate path names must be given if required. Some commands also allow directory names. In this case, all files in the directory are acted on. If the directory contains non-SCCS or unreadable files, these are ignored. A file name must not begin with a minus sign (-).

The special symbol - may be used to cause an SCCS command to read a list of file names from the standard input. These file names are then used as names for the files to be processed. The list must be terminated by the end-of-file character CONTROL-D.

Any options given with a command apply to all files. The SCCS commands process the options before any file names so the options may appear anywhere on the command line.

File names are processed left to right. If a command encounters a fatal error, it stops processing the current file and, if any other files have been given, begins processing the next.

File Administrator

Every SCCS file requires an administrator to maintain and keep the file in order. The administrator is usually the user who created the file and therefore owns the file. When other users intend to access the file, the administrator must ensure that they have adequate access. Several SCCS commands are used by the administrator to define who has access to the versions in a given s-file. These commands are described later.

Creating and Using s-files

The s-file is the key element in SCCS. It provides compact storage for all versions of a given file and automatic maintenance of the relationships between the versions.

This section explains how to use the **admin, get,** and **delta** commands to create and use s-files. In particular, it describes how to create the first version of a file, how to retrieve versions for reading and editing, and how to save new versions.

Creating an s-file

You can create an s-file from an existing text file using the **-i** (for "initialize") option of the **admin** command. The command has the form

```
admin -ifilename  s.filename
```

where **-i***filename* gives the name of the text file from which the s-file is to be created, and **s.***filename* is the name of the new s-file. The name must begin with **s.** and must be unique; no other s-file in the same directory may have the same name. For example, suppose the file named **demo.c** contains the short C language program

```
#include <stdio.h>

main ()
{
printf("This is version 1.1 \n");
}
```

To create an s-file, type

```
admin -idemo.c  s.demo.c
```

This command creates the s-file **s.demo.c** and copies the first delta describing the contents of **demo.c** to this new file. The first delta is numbered 1.1.

After creating an s-file, the original text file should be removed using the **rm** command. It is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in the next section.

When you are first creating an s-file, the **admin** command may display the warning message

```
No id keywords (cm7)
```

In general, this message can be ignored unless you have specifically included keywords in your file (see the section "Using Identification Keywords" later in this chapter).

Note that only a user with write permission in the directory containing the s-file may use the **admin** command on that file. This protects the file from administration by unauthorized users.

Retrieving a File for Reading

You can retrieve a file for reading from a given s-file by using the **get** command. The command has the form

```
get s.filename . . .
```

where **s.***filename* is the name of the s-file containing the text file. The command retrieves the lastest version of the text file and copies it to a regular file. The file has the same name as the s-file but with the **s.** removed. It also has read-only file permissions. For example, suppose the s-file **s.demo.c** contains the first version of the short C program shown in the previous section. To retrieve this program, type

```
get s.demo.c
```

The command retrieves the program and copies it to the file named **demo.c.** You may then display the file just as you do any other text file.

The command also displays a message that describes the SID of the retrieved file and its size in lines. For example, after retrieving the short C program from **s.demo.c,** the command displays the message

```
1.1
6 lines
```

You may also retrieve more than one file at a time by giving multiple s-file names in the command line. For example, the command

```
get s.demo.c  s.def.h
```

retrieves the contents of the s-files **s.demo.c** and **s.def.h** and copies them to the text files **demo.c** and **def.h.** Multiple s-file names in a command must be separated by spaces or tabs. When the **get** command displays information about the files, it places the corresponding file name before the relevant information.

Retrieving a File for Editing

You can retrieve a file for editing from a given s-file by using the **-e** (for "editing") option of the **get** command. The command has the form

```
get -e s.filename ...
```

where **s.***filename* is the name of the s-file containing the text file. You may give more than one file name if you wish. If you do, you must separate the names with spaces or tabs.

get retrieves the latest version of the text file and copies it to an ordinary text file. The file has the same name as the s-file but with the **s.** removed. It has read and write file permissions. For example, suppose the s-file **s.demo.c** contains the first version of a C program. To retrieve this program, type

```
get -e s.demo.c
```

The command retrieves the program and copies it to the file **demo.c.** You may edit the file just as you do any other text file.

If you give more than one file name, the command creates files for each corresponding s-file. Since the **-e** option applies to all the files, you may edit each one.

After retrieving a text file, the command displays a message giving the SID of the file and its size in lines. The message also displays a proposed SID, that is, the SID for the new version after editing. For example, after retrieving the six-line C program in **s.demo.c**, the command displays the message

```
1.1
new delta 1.2
6 lines
```

The proposed SID is 1.2. If more than one file is retrieved, the corresponding file name precedes the relevant information.

Note that any changes made to the text file are not immediately copied to the corresponding s-file. To save these changes you must use the **delta** command described in the next section. To help keep track of the current file version, the **get** command creates another file, called a p-file, that contains information about the text file. This file is used by a subsequent **delta** command when saving the new version. The p-file has the same name as the s-file but begins with "p.". The user must not access the p-file directly.

Saving a New Version of a File

You can save a new version of a text file by using the **delta** command. The command has the form

```
delta s.filename
```

where **s.***filename* is the name of the s-file from which the modified text file was retrieved. For example, to save changes made to a C program in the file **demo.c** (which was retrieved from the file **s.demo.c**), type

```
delta s.demo.c
```

Before saving the new version, the **delta** command asks for comments explaining the nature of the changes. It displays the prompt

```
comments?
```

You may type any text you think appropriate, up to 512 characters. The comment must end with a newline character. If necessary, you can start a new line by typing a backslash (\) followed by a newline character. If you do not wish to include a comment, just type a newline character.

Once you have given a comment, the command uses the information in the corresponding p-file to compare the original version with the new version. A list of all the changes is copied to the s-file. This is the new delta.

After a command has copied the new delta to the s-file, it displays a message showing the new SID and the number of lines inserted, deleted, or left unchanged in the new version. For example, if the C program has been changed to

```
#include <stdio.h>

main()
{
int i = 2;

printf("This is version 1.%d \n", i);
}
```

the command displays the message

```
1.2
3 inserted
1 deleted
5 unchanged
```

Once a new version is saved, the next **get** command retrieves the new version. The command ignores previous versions. If you wish to retrieve a previous version, you must use the **-r** option of the **get** command as described in the next section.

Retrieving a Specific Version

You can retrieve any version you wish from an s-file by using the **-r** (for "retrieve") option of the **get** command. The command has the form

```
get [ -e ] -rSID  s.filename ...
```

where **-e** is the edit option, -r*SID* gives the SID of the version to be retrieved, and **s.***filename* is the name of the s-file containing the file to be retrieved. You may give more than one file name. Names must be separated with spaces or tabs.

The command retrieves the given version and copies it to the file having the same name as s-file but with the **s.** removed. The file has read-only permission unless you also give the **-e** option. If multiple file names are given, one text file of the given version is retrieved from each. For example, the command

```
get -r1.1 s.demo.c
```

retrieves version 1.1 from the s-file **s.demo.c,** but the command

```
get -e -r1.1 s.demo.c s.def.h
```

retrieves for editing a version 1.1 from both **s.demo.c** and **s.def.h.** If you give the number of a version that does not exist, the command displays an error message.

You may omit the level number of a version number if you wish, that is, just give a release number. If you do, the command automatically retrieves the most recent version having the same release number. For example, if the most recent version in the file **s.demo.c** is numbered 1.4, the command

```
get -r1 s.demo.c
```

retrieves the version 1.4. If there is no version with the given release number, the command retrieves the most recent version in the previous existing release.

Changing the Release Number of a File

You can direct the **delta** command to change the release number of a new version of a file by using the **-r** option of the **get** command. In this case, the **get** command has the form

```
get -e -rrel-num    s.filename ...
```

where **-e** is the required edit option, *-rrel-num* gives the new release number of the file, and **s.***filename* gives the name of the s-file containing the file to be retrieved. The new release number must be an entirely new number, that is, no existing version may have this number. You may give more than one file name.

The command retrieves the most recent version from the s-file, then copies the new release number to the p-file. On the subsequent **delta** command, the new version is saved using the new release number and level number 1. For example, if the most recent version in the s-file **s.demo.c** is 1.4, the command

```
get -e -r2 s.demo.c
```

causes the subsequent **delta** to save a new version 2.1, not 1.5. The new release number applies to the new version only; the release numbers of previous versions are not affected. Therefore, if you edit version 1.4 (from which 2.1 was derived) and save the changes, you create a new version 1.5. Similarly, if you edit version 2.1, you create a new version 2.2.

As before, the **get** command also displays a message showing the current version number, the proposed version number, and the size of the file in lines. Similarly, the subsequent **delta** command displays the new version number and the number of lines inserted, deleted, and unchanged in the new file.

Creating a Branch Version

You can create a branch version of a file by editing a version that has been previously edited. A branch version is simply a version with an SID that contains a branch and sequence number.

For example, if version 1.4 already exists, the command

```
get -e -r1.3 s.demo.c
```

retrieves version 1.3 for editing and gives 1.3.1.1 as the proposed SID.

In general, whenever **get** discovers that you wish to edit a version that already has a succeeding version, it uses the first available branch and sequence numbers for the proposed SID. For example, if you edit version 1.3 a third time, **get** gives 1.3.2.1 as the proposed SID.

You can save a branch version just like any other version by using the **delta** command.

Retrieving a Branch Version

You can retrieve a branch version of a file by using the **-r** option of the **get** command. For example, the command

```
get -r1.3.1.1 s.demo.c
```

retrieves branch version 1.3.1.1.

You may retrieve a branch version for editing by using the **-e** option of the **get** command. When retrieving for editing, **get** creates the proposed SID by incrementing the sequence number by one. For example, if you retrieve branch version 1.3.1.1 for editing, **get** gives 1.3.1.2 as the proposed SID.

As always, the command displays the version number and file size. If the given branch version does not exist, the command displays an error message.

You may omit the sequence number if you wish. In this case, the command retrieves the most recent branch version with the given branch number. For example, if the most recent branch version in the s-file **s.def.h** is 1.3.1.4, the command

```
get -r1.3.1 s.def.h
```

retrieves version 1.3.1.4.

Retrieving the Most Recent Version

You can always retrieve the most recent version of a file by using the **-t** option with the **get** command. For example, the command

```
get -t s.demo.c
```

retrieves the most recent version from the file **s.demo.c**. You may combine the **-r** and **-t** options to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, then the command

```
get -r3 -t s.demo.c
```

retrieves version 3.5. If a branch version exists that is more recent than version 3.5 (e.g., 3.2.1.5), then the above command retrieves the branch version and ignores version 3.5.

Displaying a Version

You can display the contents of a version at the standard output by using the **-p** option of the **get** command. For example, the command

```
get -p s.demo.c
```

displays the most recent version in the s-file **s.demo.c** at the standard output. Similarly, the command

```
get -p -r2.1 s.demo.c
```

displays version 2.1 at the standard output.

The **-p** option is useful for creating g-files with user-supplied names. Since this option also directs all output normally sent to the standard output, such as the SID of the retrieved file, to the standard error output, the resulting file contains only the contents of the given version. For example, the command

```
get -p s.demo.c >version.c
```

copies the most recent version in the s-file **s.demo.c** to the file **version.c**. The SID of the file and its size are copied to the standard error output.

Saving a Copy of a New Version

The **delta** command normally removes the edited file after saving it in the s-file. You can save a copy of this file by using the **-n** option of the **delta** command. For example, the command

```
delta -n s.demo.c
```

first saves a new version in the s-file **s.demo.c** and then saves a copy of the source version in the file **demo.c.** You may then display the source file as desired, but you cannot edit it.

Displaying Helpful Information

An SCCS command displays an error message whenever it encounters an error in a file. An error message has the form

```
ERROR [ filename ]: message ( code )
```

where *filename* is the name of the file being processed, *message* is a short description of the error, and *code* is the alphanumeric error code.

You may use the error code as an argument to the **help** command to display additional information about the error. The command has the form

```
help code
```

where *code* is the error code given in an error message. The command displays one or more lines of text that explain the error and suggest a possible remedy. For example, the command

```
help co1
```

displays the message

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Use the **help** command whenever you have doubt about the meaning of an error message.

Using Identification Keywords

SCCS provides several special symbols, called identification keywords, that can be used in the text of a program or document to represent predefined values. Keywords represent values ranging from the creation date and time of a given file to the name of the module containing the keyword. When a user retrieves the file for reading, SCCS automatically replaces each keyword it finds in a given version of a file with the keyword's value.

This section explains how keywords are treated by the various SCCS commands and how you may use the keywords in your own files. Only a few keywords are described in this section. For a complete list of the keywords, see **get** in Appendix B, "Programming Commands."

Inserting a Keyword into a File

You may insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%). For example, **%I%** is the keyword representing the SID of the current version, and **%H%** is the keyword representing the current date.

When a file is retrieved for reading using the **get** command, the keywords are replaced by their current values. For example, if the **%M%, %I%,** and **%H%** keywords are used in place of the module name, the SID, and the current date in the program statement

```
char header(100) = {" %M% %I% %H% "};
```

then these keywords are expanded in the retrieved version of the program

```
char header(100) = {" MODNAME 2.3 07/07/77 "};
```

The **get** command does not replace keywords when retrieving a version for editing. SCCS assumes that you wish to keep the keywords (and not their values) when you save the new version of the file.

To indicate that a file has no keywords, the **get, delta**, and **admin** commands display the message

```
No id keywords (cm7)
```

This message is normally treated as a warning, letting you know that no keywords are present. However, you may change the operation of the system to make this a fatal error as explained later in this chapter.

Assigning Values to Keywords

The values of most keywords are predefined by the system, but some, such as the value for the **%M%** keyword, can be explicitly defined by the user. To assign a value to a keyword, you must set the corresponding s-file flag to the desired value. You can do this by using the **-f** option of the **admin** command.

For example, to set the **%M%** keyword to "cdemo", set the **m** flag as in the command

```
admin -fmcdemo s.demo.c
```

This command records "cdemo" as the current value of the **%M%** keyword. Note that if you do not set the **m** flag, SCCS uses the name of the original text file for **%M%** by default.

The **t** and **q** flags are also associated with keywords. A description of these flags and the corresponding keywords can be found in the entry **get** in Appendix B, "Programming Commands." You can change keyword values at any time.

Forcing Keywords

You can force a fatal error if a version is found to contain no keywords by setting the **i** flag in the given s-file. The flag causes the **delta** and **admin** commands to stop processing of the given version and report an error if no keywords are found. The flag is useful for ensuring that keywords are used properly in a given file.

To set the **i** flag, you must use the **-f** option of the **admin** command. For example, the command

```
admin -fi s.demo.c
```

sets the **i** flag in the s-file **s.demo.c**. Subsequent **delta** or **admin** commands that access this file print an error message if the given version does not contain keywords.

Note that if you attempt to set the **i** flag at the same time you create an s-file and if the initial text file contains no keywords, the **admin** command displays a fatal error message and stops without creating the s-file.

Using s-file Flags

An s-file flag is a special value that defines how SCCS will operate on the corresponding s-file. The s-file flags are stored in the s-file and are read by each SCCS command before it operates on the file. s-file flags affect keyword checking, keyword replacement values, and default values for commands.

This section explains how to set and use s-file flags. It also describes the action of commonly used flags. For a complete description of all flags, see the entry **admin** in Appendix B, "Programming Commands."

Setting s-file Flags

You can set the flags in a given s-file by using the **-f** option of the **admin** command. The command has the form

```
admin -fflag    s.filename
```

where **-f***flag* gives the flag to be set, and **s.***filename* gives the name of the s-file in which the flag is to be set. For example, the command

```
admin -fi s.demo.c
```

sets the **i** flag in the s-file **s.demo.c.**

Note that some s-file flags take values when they are set. For example, the **m** flag requires that a module name be given. When a value is required, it must immediately follow the flag name as in the command

```
admin -fmdmod s.demo.c
```

which sets the **m** flag to the module name "dmod".

Using the i Flag

The **i** flag causes the **admin** and **delta** commands to print a fatal error message and stop if no keywords are found in the given text file. The flag is used to prevent a file version that contains expanded keywords from being saved as a new version. (Saving an expanded version destroys the keywords for all subsequent versions.)

When the **i** flag is set, each new version of a file must contain at least one keyword. Otherwise, the version cannot be saved.

Using the d Flag

The **d** flag gives the default SID for versions retrieved by the **get** command. The flag takes an SID as its value. For example, the command

```
admin -fd1.1 s.demo.c
```

sets the default SID to 1.1. A subsequent **get** command that does not use the **-r** option will retrieve version 1.1.

Using the v Flag

The **v** flag allows you to include modification requests in an s-file. Modification requests are names or numbers that may be used as a shorthand method to indicate the reason for each new version.

When the **v** flag is set, the **delta** command asks for the modification requests just before asking for comments. The **v** flag also allows the **-m** option to be used in the **delta** and **admin** commands.

Removing an s-file Flag

You can remove an s-file flag from an s-file by using the **-d** option of the **admin** command. The command has the form

```
admin -dflag s.filename
```

where **-d***flag* gives the name of the flag to be removed and **s.***filename* is the name of the s-file from which the flag is to be removed. For example, the command

```
admin -di s.demo.c
```

removes the **i** flag from the s-file **s.demo.c**. When removing a flag that takes a value, only the flag name is required. For example, the command

```
admin -dm s.demo.c
```

removes the **m** flag from the s-file.

The **-d** option and the **-i** ("initialize") option must not be used at the same time.

Modifying s-file Information

Every s-file contains information about the deltas it contains. Normally, this information is maintained by the SCCS commands and is not directly accessible by the user. Some information, however, is specific to the user who creates the s-file and may be changed as desired to meet the user's requirements. This information is kept in two special parts of the s-file called the delta table and the description field.

The delta table contains information about each delta, such as the SID and the date and time of creation. It also contains user-supplied information, such as comments and modification requests. The description field contains a user-supplied description of the s-file and its contents. Both parts can be changed or deleted at any time to reflect changes to the s-file contents.

Adding Comments

You can add comments to an s-file by using the **-y** option of the **delta** and **admin** commands. This option causes the given text to be copied to the s-file as the comment for the new version. The comment may be any combination of letters, digits, and punctuation symbols. No embedded newline characters are allowed. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line. For example, the command

```
delta -y"George Wheeler" s.demo.c
```

saves the comment "George Wheeler" in the s-file **s.demo.c.**

The **-y** option is typically used in shell procedures as part of an automated approach to maintaining files. When the **-y** option is used, the **delta** command does not print the corresponding comment prompt, so no interaction is required. If more than one s-file is given in the command line, the given comment applies to them all.

Changing Comments

You can change the comments in an s-file by using the **cdc** command. The command has the form

```
cdc -rSID s.filename
```

where **-r***SID* gives the SID of the version with the comment to be changed, and **s.***filename* is the name of the s-file containing the version. The command asks for a new comment by displaying the prompt

```
comments?
```

You may type any sequence of characters up to 512 characters long. The sequence may contain embedded newline characters if each such newline character is preceded by a backslash (\). The sequence must be terminated with a newline character. For example, the command

```
cdc -r3.4 s.demo.c
```

prompts for a new comment for version 3.4.

Although the command does not delete the old comment, it is no longer directly accessible by the user. The new comment contains the login name of the user who invoked the **cdc** command and the time the comment was changed.

Adding Modification Requests

You can add modification requests to an s-file, when the **v** flag is set, by using the **-m** option of the **delta** and **admin** commands. A modification request is a shorthand method of describing the reason for a particular version. Modification requests are usually names or numbers that the user has chosen to represent a specific request.

The **-m** option causes the given command to save the requests following the option. A request may be any combination of letters, digits, and punctuation symbols. If spaces are used, you must enclose the request in double quotes. For example, the command

```
delta -m"error35 optimize10" s.demo.c
```

copies the requests "error35" and "optimize10" to **s.demo.c** while saving the new version.

The **-m** option when used with the **admin** command must be combined with the **-i** option. Furthermore, the **v** flag must be explicitly set with the **-f** option. For example, the command

```
admin -idef.h -m"error0" -fv s.def.h
```

inserts the modification request "error0" in the new file **s.def.h.**

The **delta** command does not prompt for modification requests if you use the **-m** option.

Changing Modification Requests

You can change modification requests, when the **v** flag is set, by using the **cdc** command. The command asks for a list of modification requests by displaying the prompt

```
MRs?
```

You may type any number of requests. Each request may have any combination of letters, digits, or punctuation symbols. No more than 512 characters are allowed and the last request must be terminated with a newline character. If you wish to remove a request, you must precede the request with an exclamation mark (!). For example, the command

```
cdc -r1.4 s.demo.c
```

asks for changes to the modification requests. The response

```
MRs? error36 !error35
```

adds the request "error36" and removes "error35".

Adding Descriptive Text

You can add descriptive text to an s-file by using the **-t** option of the **admin** command. Descriptive text is any text that describes the purpose and reason for the given s-file. Descriptive text is independent of the contents of the s-file and can only be displayed using the **prs** command.

The **-t** option directs the **admin** command to copy the contents of a given file into the description field of the s-file. The command has the form

admin -t*filename* s.*filename*

where **-t***filename* gives the name of the file containing the descriptive text, and **s.***filename* is the name of the s-file to receive the descriptive text. The file to be inserted may contain any amount of text. For example, the command

admin -tcdemo s.demo.c

inserts the contents of the file **cdemo** into the description field of the s-file **s.demo.c.**

The **-t** option may also be used when creating the s-file to initialize the description field. For example, the command

admin -idemo.c -tcdemo s.demo.c

inserts the contents of the file **cdemo** into the new s-file **s.demo.c.** If the **-t** option is not used, the description field of the new s-file is left empty.

You can remove the current descriptive text in an s-file by using the **-t** option without a file name. For example, the command

admin -t s.demo.c

removes the descriptive text from the s-file **s.demo.c.**

Printing from an s-file

This section explains how to display information contained in an s-file using the **prs** command. The command has a variety of options that control the display format and content.

Using a Data Specification

You can explicitly define the information to be printed by using the **-d** option of the **prs** command. The command copies user-specified information to the standard output. The command has the form

prs -d*spec* s.*filename*

where **-d***spec* is the data specification, and **s.***filename* is the name of the s-file from which the information is to be taken.

The data specification is a string of data keywords and text. A data keyword is an uppercase letter, enclosed in colons (:). It represents a value contained in the given s-file. For example, the keyword **:I:** represents the SID of a given version; **:F:** represents the file name of the given s-file; and **:C:** represents the comment line associated with a given version. Data keywords are replaced by these values when the information is printed.

For example, the command

```
prs -d" version: :I: filename: :F: " s.demo.c
```

may produce the line

```
version: 2.1 filename: s.demo.c
```

A complete list of the data keywords is given in the entry **prs** in Appendix B, "Programming Commands."

Printing a Specific Version

You can print information about a specific version in a given s-file by using the **-r** option of the **prs** command. The command has the form

```
prs -rSID s.filename
```

where **-r***SID* gives the SID of the desired version, and **s.***filename* is the name of the s-file containing the version. For example, the command

```
prs -r2.1 s.demo.c
```

prints information about version 2.1 in the s-file **s.demo.c.**

If the **-r** option is not specified, the command prints information about the most recently created delta.

Printing Later and Earlier Versions

You can print information about a group of versions by using the **-l** and **-e** options of the **prs** command. The **-l** option causes the command to print information about all versions immediately succeeding the given version. The **-e** option causes the command to print information about all versions immediately preceding the given version. For example, the command

```
prs -r1.4 -e s.demo.c
```

prints all information about versions that precede version 1.4 (e.g., 1.3, 1.2, and 1.1). The command

```
prs -r1.4 -l s.abc
```

prints information about versions that succeed version 1.4 (e.g., 1.5, 1.6, and 2.1).

If both options are given, information about all versions is printed.

Editing by Several Users

SCCS allows any number of users to access and edit versions of a given s-file. Since users are likely to access the s-file at the same time, the system is designed to allow concurrent editing of different versions. Normally, the system prevents concurrent editing of the same version, but you can change the operation of the system to allow concurrent editing of the same version by setting the **j** flag in the given s-file.

The following sections explain how to perform concurrent editing and how to save edited versions when you have retrieved more than one version for editing.

Editing Different Versions

SCCS allows several different versions of a file to be edited at the same time. This means a user can edit version 2.1 while another user edits version 1.1. There is no limit to the number of versions that can be edited at any given time.

When several users edit different versions concurrently, each user must begin work in his own directory. If users attempt to share a directory and work on versions from the same s-file at the same time, the **get** command will refuse to retrieve a version.

Editing a Single Version

A single version of a file can be edited by more than one user if the **j** flag is set in the s-file. The flag causes the **get** command to check the p-file and create a new proposed SID if the given version is already being edited.

You can set the flag by using the **-f** option of the **admin** command. For example, the command

```
admin -fj s.demo.c
```

sets the flag for the s-file **s.demo.c**.

When the flag is set, **get** uses the next available branch SID for each new proposed SID. For example, suppose a user retrieves for editing version 1.4 in the file **s.demo.c** and that the proposed version is 1.5. If a short time later (before the first user has saved his changes) another user retrieves version 1.4 for editing, then the proposed version for the new user will be 1.4.1.1, since version 1.5 is already proposed and likely to be taken. In no case will a version edited by two separate users result in a single new version.

Saving a Specific Version

When editing two or more versions of a file, you can direct the **delta** command to save a specific version by using the **-r** option to give the SID of that version. The command has the form

```
delta -rSID  s.filename
```

where **-r***SID* gives the SID of the version being saved and **s.***filename* is the s-file to receive the new version. The SID may be the SID of the version you have just edited, or the proposed SID for the new version. For example, if you have retrieved version 1.4 for editing (and no version 1.5 exists), both commands

```
delta -r1.5 s.demo.c
```

and

```
delta -r1.4 s.demo.c
```

save version 1.5.

Protecting s-files

SCCS uses the normal XENIX system file permissions to protect s-files from changes by unauthorized users. In addition to the XENIX system protections, SCCS provides two of its own ways to protect the s-files: the user list and the protection flags. The user list is a list of login names and group IDs of users allowed to access the s-file and create new versions of the file. The protection flags are three special s-file flags that define which versions are currently accessible to otherwise authorized users. The following sections explain how to set and use the user list and protection flags.

Adding a User to the User List

You can add a user or a group of users to the user list of a given s-file by using the **-a** option of the **admin** command. The option causes the given name to be added to the user list. The user list defines who may access and edit the versions in the s-file. The command has the form

```
admin -aname  s.filename
```

where **-a***name* gives the login name of the user or the group name of a group of users to be added to the list, and **s.***filename* gives the name of the s-file to receive the new users. For example, the command

```
admin -ajohnd -asuex -amarketing s.demo.c
```

adds the users "johnd" and "suex" and the group "marketing" to the user list of the s-file **s.demo.c.**

If you create an s-file without giving the **-a** option, the user list is left empty and all users may access and edit the files. When you explicitly give a user name or names, only those users can access the files.

Removing a User from a User List

You can remove a user or a group of users from the user list of a given s-file by using the **-e** option of the **admin** command. The option is similar to the **-a** option but performs the opposite operation. The command has the form

```
admin -ename   s.filename
```

where **-e***name* gives the login name of a user or the group name of a group of users to be removed from the list, and **s.***filename* is the name of the s-file from which the names are to be removed. For example, the command

```
admin -ejohnd -emarketing s.demo.c
```

removes the user "johnd" and the group "marketing" from the user list of the s-file **s.demo.c**.

Setting the Floor Flag

The floor flag, **f**, defines the release number of the lowest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command

```
admin -ff2 s.demo.c
```

sets the floor to release number 2. If you attempt to retrieve any versions with release numbers less than 2, an error will result.

Setting the Ceiling Flag

The ceiling flag, **c**, defines the release number of the highest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command

```
admin -fc5 s.demo.c
```

sets the ceiling to release number 5. If you attempt to retrieve any versions with release numbers greater than 5, an error will result.

Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file that are locked against further editing. You can set the flag by using the **-f** flag of the **admin** command. The flag must be followed by one or more release numbers. Multiple release numbers must be separated by commas (,). For example, the command

```
admin -fl3 s.demo.c
```

locks all versions with release number 3 against further editing. The command

```
admin -fl4,5,9 s.def.h
```

locks all versions with release numbers 4, 5, and 9.

Note that the special symbol "a" may be used to specify all release numbers. The command

```
admin -fla s.demo.c
```

locks all versions in the file **s.demo.c.**

Repairing SCCS Files

SCCS carefully maintains all SCCS files, making damage to the files very rare. Damage can result from hardware malfunctions that cause incorrect information to be copied to the file. The following sections explain how to check for damage to SCCS files and how to repair the damage or regenerate the file.

Checking an s-file

You can check a file for damage using the **-h** option of the **admin** command. The option causes the checksum of the given s-file to be computed and compared with the existing sum. An s-file's checksum is an internal value computed from the sum of all bytes in the file. If the new and existing checksums are not equal, the command displays the message

```
corrupted file (co6)
```

indicating damage to the file. For example, the command

```
admin -h s.demo.c
```

checks the s-file **s.demo.c** for damage by generating a new checksum for the file and comparing the new sum with the existing sum.

You may give more than one file name. If you do, the command checks each file in turn. You may also give the name of a directory, in which case the command checks all files in the directory.

Since failure to repair a damaged s-file can destroy the file's contents or make the file inaccessible, it is a good idea to regularly check all s-files for damage.

Editing an s-file

When an s-file is discovered to be damaged, it is a good idea to restore a back-up copy of the file from a back-up disk or tape rather than attempting to repair the file. (Restoring a back-up copy of a file is described in the *XENIX 286 System Administrator's Guide*.) If this is not possible, the file may be edited using a XENIX text editor.

To repair a damaged s-file, use the description of an s-file given in the entry **sccsfile** in "File Formats" in the *XENIX 286 C Library Guide* to locate the damaged part of the file. Use extreme care when making changes; small errors can cause unwanted results.

Changing an s-file's Checksum

After repairing a damaged s-file, you must change the file's checksum using the **-z** option of the **admin** command. For example, to restore the checksum of the repaired file **s.demo.c**, type

```
admin -z s.demo.c
```

The command computes and saves the new checksum, replacing the old sum.

Regenerating a g-file for Editing

You can create a g-file for editing without affecting the current contents of the p-file by using the **-k** option of the **get** command. The option has the same effect as the **-e** option except that the current contents of the p-file remain unchanged. The option is typically used to regenerate a g-file that has been accidentally removed or destroyed before it has been saved using the **delta** command.

Restoring a Damaged p-file

The **-g** option of the **get** command may be used to generate a new copy of a p-file that has been accidentally removed. For example, the command

```
get -e -g s.demo.c
```

creates a new p-file entry for the most recent version in **s.demo.c**. If the file **demo.c** already exists, it will not be changed by this command.

Using Other Command Options

Many of the SCCS commands provide options that control their operation in useful ways. This section describes these options and explains how you may use them to perform useful work.

Getting Help with SCCS Commands

You can display helpful information about an SCCS command by giving the name of the command as an argument to the **help** command. The **help** command displays a short explanation of the command and command syntax. For example, the command

```
help rmdel
```

displays the message

```
rmdel:
    rmdel -rSID name...
```

Creating a File with the Standard Input

You can direct **admin** to use the standard input as the source for a new s-file by using the **-i** option without a file name. For example, the command

```
admin -i s.demo.c <demo.c
```

causes **admin** to create a new s-file named **s.demo.c** using the text file **demo.c** as its first version.

This method of creating a new s-file is typically used to connect **admin** to a pipe. For example, the command

```
cat mod1.c mod2.c | admin -i s.mod.c
```

creates a new s-file **s.mod.c** that contains the first version of the concatenated files **mod1.c** and **mod2.c.**

Starting at a Specific Release

The **admin** command normally starts numbering versions with release number 1. You can direct the command to start with any given release number by using the **-r** option. The command has the form

```
admin -rrel-num s.filename
```

where **-r***rel-num* gives the value of the starting release number, and **s.***filename* is the name of the s-file to be created. For example, the command

```
admin -idemo.c -r3 s.demo.c
```

starts with release number 3. The first version is 3.1.

Adding a Comment to the First Version

You can add a comment to the first version of a file by using the **-y** option of the **admin** command when creating the s-file. For example, the command

```
admin -idemo.c -y"George Wheeler" s.demo.c
```

inserts the comment "George Wheeler" in the new s-file **s.demo.c.**

The comment may be any combination of letters, digits, and punctuation symbols. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line.

If the **-y** option is not used when creating an s-file, a comment of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically inserted.

Suppressing Normal Output

You can suppress the normal display of messages created by the **get** command by using the **-s** option. The option prevents information, such as the SID of the retrieved file, from being copied to the standard output. The option does not suppress error messages.

The **-s** option is often used with the **-p** option to pipe the output of the **get** command to other commands. For example, the command

```
get -p -s s.demo.c | lpr
```

copies the most recent version in the s-file **s.demo.c** to the line printer.

You can also suppress the normal output of the **delta** command by using the **-s** option. This option suppresses all output normally directed to the standard output, except for the normal comment prompt.

Including and Excluding Deltas

You can explicitly define which deltas you wish to include and which you wish to exclude when creating a g-file by using the **-i** and **-x** options of the **get** command.

The **-i** option causes the command to apply the given deltas when constructing a version. The **-x** option causes the command to ignore the given deltas when constructing a version. Both options must be followed by one or more SIDs. If multiple SIDs are given, they must be separated by commas (,). A range of SIDs may be given by separating two SIDs with a hyphen (-). For example, the command

```
get -i1.2,1.3 s.demo.c
```

causes deltas 1.2 and 1.3 to be used to construct the g-file. The command

```
get -x1.2-1.4 s.demo.c
```

causes deltas 1.2 through 1.4 to be ignored when constructing the file.

The **-i** option is useful if you wish to apply the same changes to more than one version. For example, the command

```
get -e -i4.1 -r3.3 s.demo.c
```

retrieves version 3.3 for editing. When the file is retrieved, the changes in delta 4.1 are automatically applied to it, making the g-file the same as if version 3.3 had been edited by hand using the changes in delta 4.1. These changes can be saved immediately by issuing a **delta** command. No editing is required.

The **-x** option is useful if you wish to remove changes performed on a given version. For example, the command

```
get -e -x1.5 -r1.6 s.demo.c
```

retrieves version 1.6 for editing. When the file is retrieved, the changes in delta 1.5 are automatically left out of it, making the g-file the same as if version 1.4 had been changed according to delta 1.6 (with no intervening delta 1.5). These changes can be saved immediately by issuing a **delta** command. No editing is required.

When deltas are included or excluded, **get** compares them with the deltas normally used in constructing the given version. If two deltas attempt to change the same line of the retrieved file, the command displays a warning message. The message shows the range of lines in which the problem may exist. Corrective action, if required, is the responsibility of the user.

Listing the Deltas of a Version

You can create a table showing the deltas required to create a given version by using the **-l** option. The option causes the **get** command to create an l-file that contains the SIDs of all deltas used to create the given version.

The option is typically used to create a history of a given version's development. For example, the command

```
get -l s.demo.c
```

creates a file named **l.demo.c** containing the deltas required to create the most recent version of **demo.c.**

You can display the list of deltas required to create a version by using the **-lp** option. This option performs the same function as the **-l** option, except that it copies the list to the standard output. For example, the command

```
get -lp -r2.3 s.demo.c
```

copies the list of deltas required to create version 2.3 of **demo.c** to the standard output.

Note that the **-l** option may be combined with the **-g** option to create a list of deltas without retrieving the actual version.

Mapping Lines to Deltas

You can map each line in a given version to its corresponding delta by using the **-m** option of the **get** command. The option causes each line in a g-file to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the beginning of the line by a tab character. The option is typically used to review the history of each line in a given version.

Naming Lines

You can name each line in a given version with the current module name (i.e., the value of the **%M%** keyword) by using the **-n** option of the **get** command. The option causes each line of the retrieved file to be preceded by the value of the **%M%** keyword and a tab character.

The **-n** option is typically used to indicate that a given line is from the given file. When both the **-m** and **-n** options are specified, each line begins with the **%M%** keyword.

Displaying a List of Differences

You can display a detailed list of the differences between a new version of a file and the previous version by using the **-p** option of the **delta** command. The option causes the command to display the differences in a format similar to the output of the XENIX **diff** command.

Displaying File Information

You can display information about a given version by using the **-g** command of the **get** option. The option suppresses the actual retrieval of a version and causes only the information about the version, such as the SID and size, to be displayed.

The option is often used with the **-r** option to check for the existence of a given version. For example, the command

```
get -g -r4.3 s.demo.c
```

displays information about version **4.3** in the s-file **s.demo.c.** If the version does not exist, the command displays an error message.

Removing a Delta

You can remove a delta from an s-file by using the **rmdel** command. The command has the form

```
rmdel -rSID s.filename
```

where -r*SID* gives the SID of the delta to be removed, and **s.***filename* is the name of the s-file from which the delta is to be removed. The delta must be the most recently created delta in the s-file. Furthermore, the user must have write permission in the directory containing the s-file and must either own the s-file or be the user who created the delta.

For example, the command

```
rmdel -r2.3 s.demo.c
```

removes delta **2.3** from the s-file **s.demo.c.**

The **rmdel** command will refuse to remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value. The command will also refuse to remove a delta that is currently being edited.

The **rmdel** command should be reserved for those cases in which incorrect, global changes were made to an s-file.

Note that **rmdel** changes the type indicator of the given delta from "D" to "R". A type indicator defines the type of delta.

Searching for Strings

You can search for strings in files created from an s-file by using the **what** command. The command searches for the symbol "#(@)" (the current value of the **%Z%** keyword) in the given file and prints on the standard output all text immediately following the symbol up to the next double quote ("), greater than (>), backslash (\), newline, or (nonprinting) NULL character. For example, if the s-file **s.demo.c** contains the line

```
char id[] = "%Z%%M%:%I%";
```

and the command

```
get -r3.4 s.prog.c
```

is executed, then the command

```
what prog.c
```

displays

```
prog.c:
        prog.c:3.4
```

You may also use **what** to search files not created by SCCS commands.

Comparing SCCS Files

You can compare two versions from a given s-file by using the **sccsdiff** command. This command prints on the standard output the differences between two versions of the s-file. The command has the form

```
sccsdiff -rSID1 -rSID2 s.filename
```

where **-r***SID1* and **-r***SID2* give the SIDs of the versions to be compared, and **s.***filename* is the name of the s-file containing the versions. The version SIDs must be given in the order in which they were created. For example, the command

```
sccsdiff -r3.4 -r5.6 s.demo.c
```

displays the differences between versions 3.4 and 5.6. The differences are displayed in a form similar to the XENIX **diff** command.

intel®

CHAPTER 6
adb: PROGRAM DEBUGGER

adb is a debugging tool for C and assembly language programs. It controls program execution and provides commands to examine and modify a program's data and text areas.

This chapter explains how to use **adb**. In particular, it explains how to

- Start the debugger.
- Display program instructions and data.
- Run, set breakpoints, or single-step a program.
- Patch program files and memory.

It also illustrates techniques for debugging C programs and explains how to display information in non-ASCII data files.

Starting and Stopping adb

adb provides a powerful set of commands to examine, debug, and repair executable binary files as well as examine non-ASCII data files. To use these commands you must invoke **adb** from a shell command line and specify the file or files you wish to debug. The following sections explain how to start **adb** and describe the types of files available for debugging.

Starting with a Program File

You can debug any executable C or assembly language program file by typing a command line of the form

```
adb [ filename ]
```

where *filename* is the name of the program file to be debugged. **adb** opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command

```
adb sample
```

prepares the program **sample** for examination and execution.

Once started, **adb** normally prompts with an asterisk (*) and waits for you to type commands. If you have given the name of a file that does not exist or is in the wrong format, **adb** will display an error message first, then wait for commands. For example, if you invoke **adb** with the command

```
adb sample
```

and the file **sample** does not exist, **adb** displays the message "adb: cannot open 'sample' "

You may also start **adb** without a file name. In this case, **adb** searches for the default file **a.out** in your current working directory and prepares it for debugging. Thus, the command

```
adb
```

is the same as typing

```
adb a.out
```

adb displays an error message and waits for a command if the **a.out** file does not exist.

Starting with a Core Image File

adb can also examine the core image files of programs that caused fatal system errors. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time of the error and provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the name of both the core file and the program file. The command line has the form

```
adb programfile corefile
```

where *programfile* is the program that caused the error, and *corefile* is the core image file generated by the system. **adb** then uses information from both files to provide responses to your commands.

If you do not give a core image file, **adb** searches for the default core file, named **core**, in your current working directory. If it is found, **adb** uses it regardless of whether or not the file belongs to the given program. You can prevent **adb** from opening this file by using the hyphen (-) in place of the core file name. For example, the command

```
adb sample -
```

prevents **adb** from searching your current working directory for a core file.

Starting adb with Data Files

You can use **adb** to examine data files by giving the name of the data file in place of the program or core file. For example, to examine a data file named **outdata,** type

```
adb outdata
```

adb opens this file and lets you examine its contents.

This method of examining files is very useful if the file contains non-ASCII data. **adb** provides a way to look at the contents of the file in a variety of formats and structures. Note that **adb** may display a warning when you give the name of a non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

Starting with the Write Option

You can make changes and corrections in a program or data file using **adb** if you open it for writing using the **-w** option. For example, the command

```
adb -w sample
```

opens the program file **sample** for writing. You may then use **adb** commands to examine and modify this file.

Note that the **-w** option causes **adb** to create a given file if it does not already exist. The option also can be used to write directly to memory after executing the given program. See the section "Patching Binary Files" later in this chapter.

Starting with the Prompt Option

You can define the prompt used by **adb** by using the **-p** option. The option has the form

```
-p prompt
```

where *prompt* is any combination of characters. If you use spaces, enclose *prompt* in quotes. For example, the command

```
adb -p "Mar 10->" sample
```

sets the prompt to "Mar 10->". The new prompt takes the place of the default prompt (*) when **adb** begins to prompt for commands.

Make sure you put at least one space between the **-p** and the new prompt, otherwise **adb** will display an error message. Note that **adb** automatically supplies a space at the end of the new prompt, so you do not have to supply one.

Leaving adb

You can stop **adb** and return to the system shell by using the **$q** or **$Q** commands. You can also stop the debugger by typing CONTROL-D.

You cannot stop **adb** by pressing the INTERRUPT or QUIT keys. These keys are caught by **adb** and cause it to to wait for a new command.

Displaying Instructions and Data

adb provides several commands for displaying the instructions and data of a given program and the data of a given data file. The commands have the form

address [, *count*] = *format*

address [, *count*] ? *format*

address [, *count*] / *format*

where *address* is a value or expression giving the location of the instruction or data item, *count* is an expression giving the number of items to be displayed, and *format* is an expression defining how to display the items. The equal sign (=), question mark (?), and slash (/) tell **adb** from what source to take the item to be displayed.

The following sections explain how to form addresses, how to choose formats, and how to use the display commands.

Forming Addresses

In **adb**, every address has the form

[*segment* :] *offset*

where *segment* is an expression giving the address of a specific segment of memory, and *offset* is an expression giving an offset from the beginning of the specified segment to the desired item. Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

segment: is optional. If not given, the most recently typed segment is used.

Forming Expressions

Expressions may contain decimal, octal, and hexadecimal integers, symbols, **adb** variables, register names, and a variety of arithmetic and logical operators.

Decimal, Octal, and Hexadecimal Integers

Decimal integers must begin with a nonzero decimal digit. Octal numbers must begin with a zero and may have octal digits only. Hexadecimal numbers must begin with the prefix "0x" and may contain decimal digits and the letters "a" through "f" (in both uppercase and lowercase). The following are valid numbers:

Decimal	Octal	Hexadecimal
34	042	0x22
4090	07772	0xffa

Although decimal numbers are displayed with a trailing decimal point (.), you must not use the decimal point when typing the number.

Symbols

Symbols are the names of global variables and functions defined within the program being debugged and are equal to the address of the given variable or function. Symbols are stored in the program's symbol table and are available if the symbol table has not been stripped from the program file (see **strip** in Appendix B, "Programming Commands").

In expressions, you may spell the symbol exactly as it is in the source program or as it has been stored in the symbol table. Symbols in the symbol table are no more than eight characters long, and those defined in C programs are given a leading underscore (_). The following are examples of symbols:

```
main    _main hex2bin  _out_of
```

Note that if the spelling of any two symbols is the same (except for a leading underscore), **adb** will ignore one of the symbols and allow references only to the other. For example, if both "main" and "_main" exist in a program, then **adb** accesses only the first to appear in the source and ignores the other.

When you use the **?** command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the command sometimes gives a function name when it displays data. This does not happen if the **?** command is used for text (instructions) and the **/** command for data. Local variables cannot be addressed.

adb Variables

adb automatically creates a set of its own variables whenever you start the debugger. These variables are set to the addresses and sizes of various parts of the program file as defined below.

d	size of data
e	entry address of the program
m	execution type
n	number of segments
t	size of text

adb reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

You can use the current value of a variable in an expression by preceding the variable name with a less-than sign (<). For example, the current value of the base variable **b** is

```
<b
```

You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater-than sign (>). The assignment has the form

expression > *variable-name*

where *expression* is the value to be assigned to the variable, and *variable-name* must be a single letter. For example, the assignment

```
0x2000>b
```

assigns the hexadecimal value "0x2000" to the variable **b**.

You can display the value of all currently defined **adb** variables by using the **$v** command. The command lists the variable names followed by their values in the current format and displays any variable with a value that is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

Current Address

adb has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The . (dot) variable, also called the current address, contains the last address to be used in a command. The " (double quotation mark) variable contains the last address to be typed with a command. The . and " variables are usually the same except when implied commands, such as the newline and caret (^) characters, are used. (These automatically increment and decrement . but leave " unchanged.)

Both . and " may be used in any expression. The less-than sign (<) is not required. For example, the command

```
=
```

displays the value of the current address and

```
"=
```

displays the last address to be typed.

Register Names

adb can use the current value of the CPU registers in expressions. You can give the value of a register by preceding its name with the less-than sign (<). **adb** recognizes the following register names:

ax	register a
bx	register b
cx	register c
dx	register d
di	data index
si	stack index
bp	base pointer
fl	status flags
ip	instruction pointer
cs	code segment
ds	data segment
ss	stack segment
es	extra segment
sp	stack pointer

For example, the value of the **ax** register can be given as

```
<ax
```

Note that register names may not be used unless **adb** has been started with a **core** file or a program is currently being run under **adb** control.

Operators

You may combine integers, symbols, variables, and register names with the following operators:

Unary

~	Not
-	Negative
*	Contents of location

Binary

+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
|	Bitwise inclusive OR
^	Modulo
#	Round up to the next multiple

Unary operators have higher precedence than binary operators. All binary operators have the same precedence. Thus, the expression

```
2*3 + 4
```

is equal to 10 and

```
4 + 2*3
```

is 18.

You can change the precedence of the operations in an expression by using parentheses. For example, the expression

```
4 + (2*3)
```

is equal to 10.

Note that **adb** uses signed 32-bit arithmetic. This means that values that exceed 2,147,483,647 (decimal) are displayed as negative values.

Note that the unary * operator treats the given address as a pointer. An expression using this operator resolves to the value pointed to by that pointer. For example, the expression

```
*0x1234
```

is equal to the value at the address 0x1234, whereas

```
0x1234
```

is just equal to 0x1234.

Choosing Data Formats

A format is a letter or character that defines how data is to be displayed. The following are the most commonly used formats:

Letter	Format
o	1 word in octal
d	1 word in decimal
D	2 words in decimal
x	1 word in hexadecimal
X	2 words in hexadecimal
u	1 word as an unsigned integer
f	2 words in floating point
F	4 words in floating point
c	1 byte as a character
s	a null terminated character string
i	machine instruction
b	1 byte in octal
a	the current symbolic address
A	the current absolute address
n	a newline
r	a blank space
t	a horizontal tab

A format may be used by itself or combined with other formats to present a combination of data in different forms.

The **d, o, x,** and **u** formats may be used to display **int** type variables and **D** and X to display **long** variables or 32-bit values. The **f** and **F** formats may be used to display single- and double-precision floating-point numbers. The **c** format displays **char** type variables and **s** is for arrays of **char** that end with a null character (null terminated strings).

The **i** format displays machine instructions in 8086/286 mnemonics. The **b** format displays individual bytes and is useful for display data associated with instructions or the high or low bytes of registers.

The **a, r,** and **n** formats are usually combined with other formats to make the display more readable. For example, the format

```
ia
```

causes the current address to be displayed after each instruction.

You may precede each format with a count of the number of times you wish it to be repeated. For example, the format

```
4c
```

displays four ASCII characters.

You can combine format requests to provide elaborate displays. For example, the command

```
<b, -1/4o4^8Cn
```

displays four octal words followed by their ASCII interpretation from the data space of the core image file. In this example, the display starts at the address "<b", the base address of the program's data. The display continues until the end of the file since the negative count "-1" causes an indefinite execution of the command until an error condition such as the end of the file occurs. In the format, "4o" displays the next four words (16-bit values) as octal numbers. The "4^" then moves the current address back to the beginning of these four words and "8C" redisplays them as eight ASCII characters. Finally, "n" sends a newline character to the terminal. The **C** format causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an "at" sign (@) followed by a lowercase letter. For example, the value 0 is displayed as "@a". The at sign itself is displayed as a double at sign, "@@".

Using the = Command

The = command displays a given address in a given format. The command is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, the command

```
main = A
```

displays the absolute address of the symbol **main** (giving the segment and offset), and the command

```
<b + 0x2000 = D
```

displays (in decimal) the sum of the variable **b** and the hexadecimal value 0x2000.

If a count is given, the same value is repeated that number of times. For example, the command

```
main, 2 = x
```

displays the value of **main** twice.

If no address is given, the current address is used instead. This is the same as the command

```
=
```

If no format is given, the previous format given for this command is used. For example, in the following sequence of commands, both **main** and **start** are displayed in hexadecimal:

```
main = x
start =
```

Using the ? and / Commands

You can display the contents of a text or data segment with the **?** and **/** commands. The commands have the form

[*address*] [, *count*] ? [*format*]

[*address*] [, *count*] / [*format*]

where *address* is an address with the given segment, *count* is the number of items you wish to display, and *format* is the format of the items you wish to display.

The **?** command is typically used to display instructions in a given text segment. For example, the command

```
main,5?ia
```

displays five instructions starting at the address **main,** and the address of each instruction is displayed immediately before it. The command

```
main,5?i
```

displays the instructions but no addresses other than the starting address.

The **/** command is typically used to check the values of variables in a program, especially variables for which no name exists in the program's symbol table. For example, the command

```
<bp-4/x
```

displays the value (in hexadecimal) of a local variable. Local variables are generally at some offset from the address pointed to by the **bp** register.

An Example: Simple Formatting

This example illustrates how to combine formats in **?** or **/** commands to display different types of values stored together in the same program. The program to be examined has the following source statements:

```
char        str1[]      = "This is a character string" ;
int         one         = 1 ;
int         number      = 456 ;
long        lnum        = 1234 ;
float       fpt         = 1.25 ;
char        str2[]      = "This is the second character string" ;

main()
{
            one = 2;
}
```

The program is compiled and stored in a file named **sample.**

To start the session, type

```
adb sample
```

You can display the value of each individual variable by giving its name and corresponding format in a **/** command. For example, the command

```
str1/s
```

displays the contents of **str1** as a string:

```
_str1: This is a character string
```

and the command

```
number/d
```

displays the contents of **number** as a decimal integer:

```
_number:    456.
```

You may choose to display a variable in a variety of formats. For example, you can display the **long** variable **lnum** as a decimal, octal, and hexadecimal number by using the commands

```
lnum/D
_lnum: 1234
lnum/O
_lnum: 02322
lnum/X
_lnum: 0x4D2
```

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, type

```
str1,5/8x
```

This command displays eight hexadecimal values on a line and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by typing

```
str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times just before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters and give an address for each line by typing

```
str1,5/4x4^8t8Cna
```

Debugging Program Execution

adb provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Note that C does not generate statement labels for programs. Thus it is not possible to refer to individual C statements when using the debugger. To use execution commands effectively, you must be familiar with the instructions generated by the C compiler and how they relate to individual C statements. One useful technique is to create an assembly language listing of your C program before using **adb**, then refer to the listing as you use the debugger. To create an assembly language listing, use the **-S** option of the **cc** command (see Chapter 2, "cc: C Compiler").

Executing a Program

You can execute a program by using the **:r** or **:R** commands. The commands have the form

[*address*] [,*count*] :r [*arguments*]

[*address*] [,*count*] :R [*arguments*]

where *address* gives the address at which to start execution, *count* is the number of breakpoints you wish to skip before one is taken, and *arguments* are the command line arguments, such as file names and options, you wish to pass to the program.

If no address is given, then the start of the program is used. Thus, to execute the program from the beginning, type

```
:r
```

If a count is given, **adb** will ignore all breakpoints until the given number have been encountered. For example, the command

```
,5:r
```

causes **adb** to skip the first five breakpoints.

If arguments are given, they must be separated by at least one space each. The arguments are passed to the program in the same way the system shell passes command line arguments to a program. You may use the shell redirection symbols if you wish.

The **:R** command passes the command arguments through the shell before starting program execution. This means you can use shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The command is especially useful if the program expects multiple file names. For example, the command

```
:R [a-z]*.s
```

passes the argument "[a-z]*.s" to the shell, where it is expanded to a list of the corresponding file names before being passed to the program.

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program you may have been running.

Setting Breakpoints

You can set a breakpoint in a program by using the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the form

address [, *count*] :br *command*

where *address* is a valid instruction address, *count* is a count of the number of times you wish the breakpoint to be skipped before it causes the program to stop, and *command* is the **adb** command you wish to execute when the breakpoint is taken.

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. For example, the command

```
main:br
```

sets a breakpoint at the start of the function **main.** The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is typically used within a function that is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint allows the program to continue to execute until the given function or instructions have been executed the specified number of times. For example, the command

```
light,5:br
```

sets a breakpoint at the fifth invocation of the function **light.** The breakpoint does not stop the program until function **light** has been called five times.

Note that no more than 16 breakpoints at a time are allowed.

Displaying Breakpoints

The command **$b** displays the location and count of each currently defined breakpoint. Breakpoints are listed by address, along with any count and/or command associated with them.

Continuing Execution

You can continue the execution of a program after it has been stopped by a breakpoint by using the **:co** command. The command has the form

[*address*] [,*count*] :co [*signal*]

where *address* is the address of the instruction at which you wish to continue execution, *count* is the number of breakpoints you wish to ignore, and *signal* is the number of the signal to send to the program (see **signal** in "System Functions" in the *XENIX 286 C Library Guide*).

If no address is given, the program starts at the next instruction after the breakpoint. If *count* is given, **adb** ignores the first *count* breakpoints.

Stopping a Program with Interrupt and Quit

You can stop execution of a program at any time by pressing the INTERRUPT or QUIT keys. These keys stop the current program and return control to **adb.** These keys are especially useful for programs with infinite loops or other program errors.

Note that whenever you press the INTERRUPT or QUIT key to stop a program, **adb** automatically saves the signal and passes it to the program if you start it again by using the **:co** command. This is very useful if you wish to test a program that uses these signals as part of its processing.

If you wish to continue execution of the program but do not wish to send the signals, type

```
:co 0
```

The command argument 0 prevents a pending signal from being sent to the program.

Single-Stepping a Program

You can single-step a program, i.e., execute it one instruction at a time, by using the **:s** command. The command executes an instruction and returns control to **adb.** The command has the form

```
[ address ][, count ]:s
```

where *address* is the address of the instruction you wish to execute, and *count* is the number of times you wish to repeat the command.

If no address is given, **adb** uses the current address. If *count* is given, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the command

```
main,5:s
```

executes the first five instructions in the function **main.**

Killing a Program

You can kill the program you are debugging by using the **:k** command. The command kills the process created for the program and returns control to **adb.** The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

Deleting Breakpoints

You can delete a breakpoint from a program by using the **:dl** command. The command has the form

address :dl

where *address* is the address of the breakpoint you wish to delete.

The **:dl** command is typically used to delete breakpoints you no longer wish to use. The following command deletes the breakpoint set at the start of the function **main:**

main:dl

Displaying the C Stack Backtrace

You can trace the path of all active functions by using the **$c** command. The command lists the names of all functions that have been called and have not yet returned control, as well as the address from which each function was called and the arguments passed to it.

For example, the command

$c

displays a backtrace of the C language functions called.

By default, the **$c** command displays all calls. If you wish to display just a few, you must supply a count of the number of calls you wish to see. For example, the command

,25$c

displays up to 25 calls in the current call path.

Note that function calls and arguments are put on the stack after the function has been called. If you put breakpoints at the entry point to a function, the function will not appear in the list generated by the **$c** command. You can remedy this problem by placing breakpoints a few instructions into the function.

Displaying CPU Registers

You can display the contents of all CPU registers by using the **$r** command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter and the instruction at the current address. The display has the form

```
ax              0x0         fl              0x0
bx              0x0         ip              0x0
cx              0x0         cs              0x0
dx              0x0         ds              0x0
di              0x0         ss              0x0
si              0x0         es              0x0
sp              0x0         sp              0x0
0:0:            addb  al,bl
```

The value of each register is given in the current default format.

Displaying External Variables

You can display the values of all external variables in the program by using the **$e** command. External variables are the variables in your program that have global scope or have been defined outside of any function. This may include variables defined in library routines used by your program.

The **$e** command is useful whenever you need a list of the names for all available variables or to quickly summarize their values. The command displays one name on each line with the variable's value (if any) on the same line.

The display has the form

```
fac:            0
_errno:         0
_end:           0
_sobuf:         0
_obuf:          0
_lastbu:        0406
_sibuf:         0
_stkmax:        0
lscadr:         02
_iob:           01664
_edata:         0
_
```

An Example: Tracing Multiple Functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements.

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x + 1;
        hr = x-y + 1;
        hcnt + + ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p + 1;
        gcnt + + ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a + 2*b;
        fr = a + b;
        fcnt + + ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

The program is compiled and stored in the file named **sample**. To start the session, type

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. You can do this with the **:br** command. For example, to set a breakpoint at the start of the function **f**, type

```
f:br
```

You can use similar commands for the **g** and **h** functions. Once you have created the breakpoints, you can display their locations by typing

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays

```
breakpoints
count       bkpt        command
1           _f
1           _g
1           _h
```

The next step is to display the first five instructions in the **f** function. Type

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address. The instructions in 8086/286 mnemonics are

```
_f:             push    bp
_f+1.:          mov     bp,sp
_f+3.:          push    di
_f+4.:          push    si
_f+5.:          call    chkstk
_f+8.:
```

You can display five instructions in **g** without their addresses by typing

```
g,5?i
```

In this case, the display is

```
_g:     push    bp
        mov     bp,sp
        push    di
        push    si
        call    chkstk
```

To start program execution, type

```
:r
```

adb displays the message

```
sample: running
```

and begins to execute. As soon as **adb** encounters the first breakpoint (at the beginning of function **f**), it stops execution and displays the message

```
breakpoint  _f:  push bp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing

```
f:dl
```

and continue the program by typing

```
:co
```

adb displays the message

```
sample: running
```

and starts the program at the next instruction. Execution continues until the next breakpoint where **adb** displays the message

```
breakpoint  _g:  push  bp
```

You can now trace the path of execution by typing

```
$c
```

The command shows that only two functions are active: **main** and **f.**

```
_f(1.,1.)          from _main + 6.
_main  (1.,470.)   from _start + 114.
```

Although the breakpoint has been set at the start of function **g,** it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type

```
,5:s
```

adb single-steps the first five instructions. Now you can list the backtrace again. Type

```
$c
```

This time the list shows three active functions:

```
_g(2.,3.)          from _f + 48.
_f(1.,1.)          from _main + 6.
_main(1.,470.)     from _start + 114.
```

You can display the contents of the integer variable **fcnt** by typing

```
fcnt/d
```

This command displays the value of **fcnt** found in memory. The number should be 1.

You can continue execution of the program and skip the first ten breakpoints by typing

```
,10:co
```

adb starts the program and displays the running message again. It does not stop the program until exactly ten breakpoints have been encountered. It then displays the message

```
breakpoint   _g:   push   bp
```

To show that these breakpoints have been skipped, you can display the backtrace again using **$c.**

```
_f    (2., 11.)     from _h + 46:
_h    (10., 9.)     from _g + 48:
_g    (11., 20.)    from _f + 48:
_f    (2., 9.)      from _h + 46:
_h    (8., 7.)      from _g + 48:
_g    (9., 16.)     from _f + 48:
_f    (2., 7.)      from _h + 46:
_h    (6., 5.)      from _g + 48:
_g    (7., 12.)     from _f + 48:
_f    (2., 5.)      from _h + 46:
_h    (4., 3.)      from _g + 48:
_g    (5., 8.)      from _f + 48:
_f    (2., 3.)      from _h + 46:
_h    (2., 1.)      from _g + 48:
```

Using the adb Memory Maps

adb prepares a set of maps for the text and data segments in your program and uses these maps to access items that you request for display. The following sections describe how to view these maps and how they are used to access the text and data segments.

Displaying the Memory Maps

You can display the contents of the memory maps by using the **$m** command. The command has the form

```
$m [ segment ]
```

where *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory.

If you have started **adb** but have not executed the program, the **$m** command display has the form

```
Text Segments
Seg #       File Pos    Phys Size    'sample' -File
63.         32.         2048.
71.         2080.       656.

Data Segments
Seg #       File Pos    Phys Size    'core' -File
39.         2736.       242.
```

Each entry gives the segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The file names to the right of the display are the program and core file names.

If you have executed the program, the command display has the form

```
Text Segments
Seg #       File Pos    Vir Size     'sample' - Memory
63.         32.         2048.
71.         2080.       656.

Data Segments
Seg #       File Pos    Vir Size     'sample' - Memory
39.         2736.       456.
```

where virtual size is the number of bytes the segment occupies in memory. This size is sometimes different than the size of the segment in the file and will often change as you execute the program. This is due to expansion of the stack or allocation of additional memory during program execution. The file names to the right always name the program file. The file position value is ignored.

If you give a segment number with the command, **adb** displays information only about that segment. For example, the command

```
$m 63
```

displays a map for segment 63 only. The display has the form

```
Segment # = 63.
Type = Text
File position = 32.
Physical Size = 2048.
```

Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the form

?m *segment-number file-position size*

and

/m *segment-number file-position size*

where *segment-number* gives the number of the segment map you wish to change, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. **?m** assigns values to a text segment entry and **/m** to a data segment entry.

For example, the following command changes the file position for segment 63 in the text map to 0x2000:

```
?m 63 0x2000
```

The command

```
/m 39 0x0
```

changes the file position for segment 39 in the data map to 0.

Creating New Map Entries

You can create new segment maps and add them to your memory map by using the **?M** and **/M** commands. Unlike **?m** and **/m**, these commands create a new map instead of changing an existing one. These commands have the form

?M *segment-number file-position size*

and

/M *segment-number file-position size*

where *segment-number* gives the number of the segment map you wish to create, *file-position* gives the offset in the file to the beginning of the given address, and *size* gives the segment size in bytes. **?M** creates a text segment entry and **/M** creates a data segment entry. The segment number must be unique. You cannot create a new map entry that has the same number as an existing one.

The **?M** and **/M** commands are especially useful if you wish to access segments otherwise allocated to your program. For example, the command

```
?M 71 0 2504
```

creates a text segment entry for segment 71 with size 2504 bytes.

Validating Addresses

Whenever you use an address in a command, **adb** checks the address to make sure it is valid. **adb** uses the segment number, file position, and size values in each map entry to validate the addresses. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the **?** command must appear in the text segments map; segments used with the **/** command must appear in the data segments map. If the value does not belong to the map, **adb** displays a bad segment error.

The next step is to check the offset to see if it is in range. The offset must be within the range

```
0 <= offset <= segment-size
```

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address as follows:

```
effective-file-address = offset + file-position
```

then uses this effective address to read from the corresponding file.

Miscellaneous Features

The following sections explain how to use a number of useful miscellaneous commands and features of **adb.**

Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored.

One typical combination is to place a **?** command after an **l** command. For example, the commands

```
?l 'Th'; ?s
```

search for and display a string that begins with the characters "Th".

Creating adb Scripts

You can direct **adb** to read commands from a text file instead of the keyboard by redirecting **adb**'s standard input file at invocation. To redirect the standard input, use the standard redirection symbol < and supply a file name. For example, to read commands from the file **script**, type

```
adb sample <script
```

The file you supply must contain valid **adb** commands. Such files are called script files and can be used with any invocation of the debugger.

Reading commands from a script file is very convenient when you wish to use the same set of commands on several different object files. Scripts are typically used to display the contents of core files after a program error. For example, a file containing the following commands can be used to display most of the relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

Setting Output Width

You can set the maximum width (in characters) of each line of output created by **adb** by using the **$w** command. The command has the form

n$w

where *n* is an integer number giving the width in characters of the display. You may give any width that is convenient for your given terminal or display device. The default width when **adb** is first invoked is 80 characters.

The command is typically used when redirecting output to a printer or special terminal. For example, the command

```
120$w
```

sets the display width to 120 characters, a common maximum width for printers.

Setting the Maximum Offset

adb normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data you are viewing with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason **adb** lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **$s** command. The command has the form

```
n$s
```

where *n* is an integer giving the new offset. For example, the command

```
4095$s
```

increases the maximum possible offset to 4095. All instructions and data no more than 4095 bytes away are given symbolic addresses.

Note that you can disable all symbolic addressing by setting the maximum offset to zero. All addresses will be given numeric values instead.

Setting Default Input Format

You can set the default format for numbers used in commands with the **$d** (decimal), **$o** (octal), and **$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with "0" or "0x" and how to display numbers when no specific format is given.

The commands are useful if you wish to work with a combination of decimal, octal, and hexadecimal numbers. For example, if you use

```
$x
```

you may give addresses in hexadecimal without prepending each address with "0x". Furthermore, **adb** displays all numbers in hexadecimal except those specifically requested to be in some other format.

When you first start **adb**, the default format is decimal. You may change this at any time and restore it as necessary using the **$d** command.

Using XENIX Commands

You can execute XENIX commands without leaving **adb** by using the **adb** escape command **!**. The escape command has the form

```
! command
```

where *command* is the XENIX command you wish to execute. The command must have any required arguments. **adb** passes this command to the system shell, which executes it. When finished, the shell returns control to **adb.**

For example, to display the date, type

```
! date
```

The system displays the date at your terminal and returns control to **adb.**

Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the = command. The command directs **adb** to display the value of an expression in a given format.

The command is often used to convert numbers in one base to another, to double-check the arithmetic performed by a program, and to display complex addresses in simpler form. For example, the command

```
0x2a = d
```

displays the hexadecimal number 0x2a as the decimal number 42, but

```
0x2a = c
```

displays it as the ASCII character "*". Expressions in a command may have any combination of symbols and operators. For example, the command

```
<d0-12*<d1 + <b + 5 = X
```

computes a value using the contents of the d0 and d1 registers and the **adb** variable **b.** You may also compute the value of external symbols as in the command

```
main + 5 = X
```

This is helpful if you wish to check the hexadecimal value of an external symbol address.

Note that the = command can also be used to display literal strings at your terminal. This is especially useful in **adb** scripts where you may wish to display comments about the script as it performs its commands. For example, the command

```
=3n"C Stack Backtrace"
```

spaces three lines, then prints the message "C Stack Backtrace" on the terminal.

An Example: Directory and Inode Dumps

This example illustrates how to create **adb** scripts to display the contents of a directory file and the inode map of a XENIX file system. The directory file is assumed to be named **dir** and contains a variety of files. The XENIX file system is assumed to be associated with the device file **/dev/src** and has the necessary permissions to be read by the user.

To display a directory file, you must create an appropriate script, e.g., in a file named **script.** Then start **adb** with the name of the directory, redirecting its input to the script.

A directory file normally contains one or more entries. Each entry consists of an unsigned "inumber" and a 14-character file name. You can display this information by adding the command

```
0,-1?ut14cn
```

to the script file. This command displays one entry for each line, separating the number and file name with a tab. The display continues to the end of the file. If you place the command

```
="inumber"8t"Name"
```

at the beginning of the script, **adb** will display the strings as headings for each column of numbers.

Once you have the script file, type

```
adb dir - <script
```

(The hyphen (-) is used to prevent **adb** from attempting to open a core file.) **adb** reads the commands from the script and the resulting display has the form

```
inumber     name
652         .
82          ..
5971        cap.c
5323        cap
0           pp
```

To display the inode table of a file system, you must create a new script, then start **adb** with the file name of the block device that contains the file system (e.g., the hard disk drive).

The inode table of a file system has a very complex structure. Each entry contains: a word value for the file's status flags; a byte value for the number of links; two byte values for the user and group IDs; a byte and word value for the size; eight word values for the location on disk of the file's blocks; and two word values for the creation and modification dates. The inode table starts at the address 02000. You can display the first entry by typing

```
02000,-1?on3bnbrdn8un2Y2na
```

Several newlines are inserted within the display to make it easier to read.

To use the script on the inode table of **/dev/src**, type

```
adb /dev/src -<script
```

(Again, the hypen (-) is used to prevent an unwanted core file.) Each entry in the display has the form

```
02000:  073145
        0163    0164    0141
        0162    10356
        28770   8236    25956   27766   25455   8236    25956   25206
        1976 Feb 5 08:34:56  1975 Dec 28 10:55:15
```

Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by using the **w** and **W** commands and invoking **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

Locating Values in a File

You can locate specific values within a file by using the **l** and **L** commands. The commands have the form

```
[ address ] ?l   value
```

where *address* is the address at which to start the search, and *value* is the value (given as an expression) to be located. The **l** command searches for two-byte values; **L** for four-byte values.

The

```
?l
```

command starts the search at the current address and continues until the first match or the end of the file. If the value is found, the current address is set to that value's address. For example, the command

```
?l 'Th'
```

searches for the first occurrence of the string value "Th". If the value is found at **main**+210, the current address is set to that address.

Writing to a File

You can write to a file by using the **w** and **W** commands. The commands have the form

[*address*] ?w *value*

where *address* is the address of the value you wish to change, and *value* is the new value. The **w** command writes two-byte values; **W** writes four-byte values. For example, the following commands change the word "This" to "The ".

```
?I 'This'
?W 'The '
```

Note that **W** is used to change all four characters.

Making Changes to Memory

You can also make changes to memory whenever a program has been executed. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program's data as it runs, for example, to temporarily change the value of program flags or constants.

intel®

CHAPTER 7
as: ASSEMBLER

This chapter describes the usage and input syntax of the XENIX 8086/286 assembler, **as.** The assembler produces relocatable object files from 8086/286 assembly language source files. Object files contain relocation information and a complete symbol table and may be linked to other object files using the XENIX link editor **ld.**

as is designed to be used in those rare cases where C programs do not satisfy a programming requirement. Thus, you can combine **as** object files with object files produced by the XENIX C compiler, **cc,** to make complete programs. Note that the output format of **as** has been designed so that if a file contains no unresolved references to external symbols, it is executable without further processing.

This chapter does not teach assembly language programming, nor does it give a detailed description of 8086/286 operation codes.

Command Usage

as is invoked as follows:

as [*option*]... *filename* ...

where each *option* is an assembler option and *filename* is the name of the assembler source file. If the file name does not have the extension ".s", **as** displays a warning message before assembling the file. Although **as** has a large number of options, the most commonly used are the **-l** and **-o** options.

The **-l** option causes the assembler to create an assembly listing that includes the source, the assembled (binary) code, and any assembly errors. The listing file is named *filename*.**L.**

The **-o** option causes the output to be placed in a given file. The option has the form

-o *outfile*

where *outfile* is the name of the file to receive the assembled program. If you do not use the **-o** option, **as** copies the output to *filename*.**o** in the current directory.

For a complete description of all assembler options, see **as** in Appendix B, "Programming Commands."

Lexical Conventions

This section describes **as** lexical conventions for identifiers, constants, white space, and comments.

Identifiers

An identifier consists of a sequence of alphanumeric characters, including periods (.) and underscores (_). The first character must not be numeric. By convention, the first eight characters are significant, but you can also define the maximum number of significant characters by using the **-nl** option. Uppercase and lowercase letters are considered distinct in identifiers.

Constants

A hex constant consists of a slash character (/) followed by a sequence of digits and one of the letters "a", "b", "c", "d", "e", or "f", any of which may be capitalized.

A decimal constant consists simply of a sequence of digits. The constant should be representable in 15 bits, i.e., be less than 32,768.

A character constant consists of one or two characters enclosed in single quotation marks ('). If a single quotation mark is used in a constant, it must be given twice to represent a single occurrence (').

The following are examples of constants:

Decimal	Hexadecimal	Character
10	/1b	'a'
32767	/7fff	'in'

White Space

Blank and tab characters may be freely interspersed between tokens but may not be used within tokens (except in character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

Comments

The vertical bar (|) introduces a comment, which extends to the end of the line where it appears. Comments are ignored by the assembler.

Assembly Segments

as assembles instruction and data statements in three segments, text, data, and bss. Segments allow division of instructions and data into separate physical segments in memory. A location counter keeps the current address within each segment during assembly and provides reference to the current instruction and data.

Text, Data, and Bss Segments

Every program is divided into at most three distinct segments of assembled code and data: the text segment, the data segment, and the bss segment. Each segment is reserved for a specific type of storage and receives different treatment from the assembler and from the XENIX linker when the final program is created.

The text segment is normally reserved for instructions but may also be used for data. Instructions in this segment are assembled, and the code is copied to the output file. Data definitions in this segment are also assembled and copied; the code is the value of the data item. The assembler does not separate the instruction and data code. If the instructions and data definitions are mixed within the source file, the resulting code is mixed within the output file.

The data segment is reserved for data. The code is copied to a different part of the output file and receives different treatment from the XENIX linker.

The bss segment is reserved for uninitialized data only. Instructions or data definitions with initial values must not be given in this segment. The assembler counts the number of bytes allocated for this segment and copies this count to the output file. It does not generate code.

The text segment is implicitly defined at the start of every assembly. Thus, any instructions or data definitions given when no other segment is explicitly defined are copied to the text segment. To start a data or bss segment, you must use a **.data** or **.bss** directive. You can explicitly start the text segment with the **.text** directive (see the section "Segment Directives," later in this chapter).

Unless otherwise specified, the first statement in the text segment is considered the program's entry point. In shared-text programs, the instructions and data in the text segment are write-protected; in nonshared-text programs, they are not. Instructions and data in the data segment are never write-protected. The bss segment is actually an extension of the data segment. It begins immediately after the data segment and is initialized to 0 at the start of program execution.

The Location Counter

The special symbol "dot" (.) is the location counter. Its value at any time is the offset from the current statement to the start of the current segment. Thus, it may be used in any statement to refer to the current location.

The location counter actually has three different offsets, one for each type of segment. Only the offset of the current segment is ever accessible. The assembler increments the current offset after it processes each statement. It increments the offset by the number of bytes in the assembled code or allocated storage.

The location counter can be assigned an explicit value if desired. Its value must not be decreased. If it is explicitly increased, the assembler generates enough null bytes of code to fill the gap between the last offset and the new offset.

Statements

A source program is composed of a sequence of statements. Statements are separated by newline characters. There are four kinds of statements:

- Null statements
- Expression statements
- Assignment statements
- Keyword statements

The format for most 8086/286 assembly language source statements is

[*labelfield*] *op-code* [*operand-field*][*comment*]

Any kind of statement may be preceded by one or more labels.

Labels

There are two kinds of labels: name labels and numeric labels. A name label consists of an identifier followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the value assigned changes the definition of the label.

A numeric label consists of a string of the digits 0 to 9 followed by a dollar sign ($) followed by a colon (:). Such a label serves to define local symbols of the form

n$

where *n* is the digit of the label. The scope of the numeric label is the labeled block in which it appears. As an example, the label "9$" is defined only between the labels **label1** and **label2:**

```
label1:
9$:         .byte 0
            .
            .
            .
label2:     .word a
```

As in the case of name labels, a numeric label assigns the current value and type of dot to the symbol.

Null Statements

A null statement is an empty statement (which may, however, have labels and a comment). A null statement is ignored by the assembler. Common examples of null statements are empty lines or comment lines.

Expression Statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its value and places it in the output stream, together with the appropriate relocation bits.

Assignment Statements

An assignment statement consists of an identifier, an equal sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of an expression is lost across an assignment. Thus you cannot declare a global symbol by assigning to it. Nor can you define a symbol to be offset from a nonlocally-defined global symbol.

As mentioned, you can assign the location counter. It is required, however, that the type of the expression assigned be of the same type as dot, and an assignment cannot decrease the value of dot. In practice, the most common assignment to dot has the form

. = . + *n*

for some number *n*; this has the effect of generating *n* null bytes.

Keyword Statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler. The syntax of the remainder depends on the keyword. All the keywords are listed in the section "Mnemonic List" later in this chapter.

Expressions

An expression is a sequence of symbols representing a value. An expression contains identifiers, constants, and operators. Each expression has a type.

Arithmetic is two's complement. All operators have equal precedence, and expressions are evaluated strictly left to right.

Expression Operators

The operators are

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Modulo
&	Logical AND
~	Logical NOT
>>	Right Shift
<<	Left Shift

Types

The assembler deals with expressions, each of which may be of a different type. Most types are attached to the keywords and are used to select the routine that treats that keyword. The types likely to be met explicitly are

undefined Upon first encounter, each symbol is undefined. A defined symbol may become undefined if it is assigned an undefined expression.

undefined external
A symbol declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor **ld** must be used to link the assembler's output with another routine that defines the undefined reference.

absolute
: An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link editor to the output file.

text
: The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value, since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of dot is text 0.

data
: The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link editor run, since previously linked programs may also have data segments. After the first **.data** statement, the value of dot is data 0.

bss
: The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously linked programs may also have bss segments. After the first **.bss** statement, the value of dot is bss 0.

external absolute, text, data, or bss
: Symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl.** However, their value and type are available to the link editor so that the program may be linked with others that reference these symbols.

other types
: Each keyword known to the assembler has a type used to select the routine that processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

Type Propagation in Expressions

When operands are combined by expression operators, the result has a type that depends on the types of the operands and on the operator. The rules involved are complex but are intended to be sensible and predictable. For purposes of expression evaluation, the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are as follows:

- If one of the operands is undefined, the result is undefined.
- If both operands are absolute, the result is absolute.
- If an absolute is combined with one of the other types mentioned above, the result has the other type.
- If two operands of other type are combined, the result has the numerically larger type.
- An other type combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are

+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand), or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is undefined external, the second must be absolute. All other combinations are illegal.

others
It is illegal to apply these operators to any but absolute symbols.

Assembler Directives

as supports a number of assembler directives (sometimes called "Pseudo-operations"). The directives modify the location counter, define the start of program segments, generate initialized data, allocate storage space, assign the global attribute to labels or symbols, and perform a variety of other tasks.

The following sections describe the directives and illustrate their use.

Even Directive

```
.even
```

The **.even** directive conditionally increments the location counter. If the location counter is odd, it is incremented by one so the next statement will be assembled at a word boundary. This is useful for forcing storage allocation to be on a word boundary after a **.byte** or **.ascii** directive.

Floating-Point Directives

```
.float    float
.double   float
```

The **.float** and **.double** directives accept one or more floating-point numbers as operands and allocate storage for each number. A floating-point number has the form

```
[-] integer.fraction [E[-] exponent ]
```

where *integer* is a decimal number, *fraction* is a combination of decimal digits, and *exponent* is an decimal number. Examples of these directives are

```
.float    25.1
.double   1.03E12
.float    -34718.235E4
```

The **.float** and **.double** directives allocate a different number of bytes. The **.float** sets aside four bytes, while **.double** sets aside eight.

Global Directive

```
.globl name ...
```

The **.globl** directive makes the text or data associated with *name* globally known to all files in a program. *name* must be explicitly defined by assignment or by appearance as a label in exactly one file. All other files that wish to access this name must use **.globl** to give the name global meaning; no other definition is allowed in these files. The link editor **ld** resolves all global references to *name* when the final program is created. If more than one name is given, they must be separated with commas (,).

Segment Directives

```
.text
.data
.bss
```

The **.text, .data,** and **.bss** directives cause the assembler to copy subsequent instruction or data code (or allocated storage) to the text, data, or bss segment respectively. The offset of the location counter is set to the previous value for that segment and subsequent statements are processed as defined in the earlier section "Assembly Segments."

The directives may be used any number of times within a program. The offset for each segment is initially set to 0. Changing a segment causes the current offset to be saved. Restoring a segment causes the old offset to be restored. Thus, each segment is copied as a contiguous block even if the original source statements were not contiguous.

Instructions and data definitions with initial values must not be used after a **.bss** directive, but symbols may be defined and dot moved by assignment.

If no explicit segment directive is given in a program, code is copied to the text segment.

Common Directive

.comm *name* [, *expression*]

The **.comm** directive makes *name* globally known to all files of the program. If *name* also appears in an assignment or as a label in a file, then the **.comm** directive has the same effect as the **.globl** directive. In this case, any *expression* given is ignored. If *name* does not have an explicit definition, then **.comm** directs the XENIX linker to automatically allocate *expression* bytes for *name* in the bss segment of the program. These bytes appear before any bytes specifically allocated within the bss segment.

Insert Directive

.insrt "*filename*"

The **.insrt** directive directs the assembler to suspend processing of the current file until all statements in the given file have been read. *filename* must be enclosed within double quotation marks. If the file cannot be opened or does not exist, the assembler displays the message

Cannot open insert file

Otherwise, it reads the contents of the file. The file may contain other **.insrt** directives; up to ten levels of directives may be nested in this way.

The **.insrt** directive is useful for including a standard set of comments or symbol assignments at the beginning of a program, e.g., the definitions for system calls found in the file **/usr/include/sys.s.** The directive is also useful for breaking up a large source program into easily manageable pieces.

ASCII Directives

.ascii */string/*
.asciz */string/*

The **.ascii** and **.asciz** directives translate *string* into an equivalent sequence of ASCII byte values and copy these bytes to allocated storage in the current segment. The **.asciz** directive also appends a null byte to the end of the sequence.

The string may contain any character in the character set except a newline. If necessary, the escape sequence "\n" may be used in place of a newline. The string must be enclosed within slashes (/) or within any character not used in the string. Examples:

```
.ascii  /"hello there"/
.ascii  "Warning- 07 07 \n"
.asciz  *abcdefg*
```

The **.asciz** directive is especially useful because some system calls and many library routines require null-terminated strings as arguments. Strings are normally null-terminated in C programs, and any string constant in a C program is stored with a terminating null, appended by the compiler.

Listing Directives

```
.list
.nlist
```

The **.list** and **.nlist** directives control output to the assembler listing file created by using the **-l** option. If **.list** is given, subsequent statements are passed to the listing file as well as being processed. If **.nlist** is given, statements are processed but not passed to the listing file.

The directives may be used any number of times to turn listing on and off. This is particularly useful when certain portions of the assembly output are not desired on a printed listing.

Block Directives

```
.blkb    [ expression ]
.blkw    [ expression ]
```

The **.blkb** and **.blkw** directives reserve blocks of storage where a block contains *expression* bytes (for **.blkb**) or *expression* words (for **.blkw**). If no expression is given, 1 is assumed. The expression must be absolute and defined during pass 1.

Note that the statement

```
. = . + expression
```

may also be used to reserve blocks of storage. In this case, the block contains *expression* bytes.

Initial Value Directives

.byte [*expression*]...
.word [*expression*]...

The **.byte** and **.word** directives reserve storage and initialize this storage to the value given by the expressions. The **.byte** directive reserves one byte for each expression and initializes that byte to the low-order byte of the expression. The **.word** directive reserves one word for each expression and initializes that word to the value of *expression*. When more than one expression is given, they must be separated by commas (,).

End Directive

.end [*expression*]

The **.end** directive marks the physical end of the source program. If *expression* is given, it indicates the entry point of the program, i.e., the starting point for execution. Otherwise, the entry point is taken to be the start of the text segment.

Note that inserted files that contain an **.end** directive terminate assembly of the entire program as well as the inserted portion.

Machine Instructions

This section presents a description of the 8086/286 instructions used by the XENIX 8086/286 assembler. This assembler supports those instructions common to the 8086 and 80286 processors, i.e., all 8086 instructions. This assembler does not support those instructions that are specific to the 80286 assembler. 80286-specific instructions can be defined by users either in numeric form or with a macro processor.

Mnemonic List

This section contains a list of the instruction mnemonics (instruction names) used by the 8086/286 assembler **as**. Some of these mnemonics are different from those used by other 8086/286 assemblers. Mnemonics marked with an asterisk may be specific to this assembler.

Mnemonic	Description	XENIX-specific
aaa	ASCII adjust for addition	
aad	ASCII adjust for division	
aam	ASCII adjust for multiplication	
aas	ASCII adjust for subtraction	
adc	add with carry	
adcb	add byte with carry	*
add	add	
addb	add byte	*
and	logical AND	
andb	logical AND byte	*
beq	long branch equal	*
bge	long branch greater or equal	*
bgt	long branch greater	*
bhi	long branch on high	*
bhis	long branch high or same	*
ble	long branch less than or equal	*
blo	long branch on low	*
blos	long branch low or same	*
blt	long branch less than	*
bne	long branch not equal	*
br	long branch	*
call	intrasegment call	
calli	intersegment call	*
cbw	convert byte to word	
clc	clear carry flag	
cld	clear direction flag	
cli	clear interrupt flag	
cmc	complement carry flag	
cmp	compare	
cmpb	compare byte	*
cmps	compare string	*
cmpsb	compare string byte	*
cwd	covert word to double word	
daa	decimal adjust for addition	
das	decimal adjust for subtraction	
dec	decrement by one	
decb	decrement byte by one	*
div	division unsigned	
divb	division unsigned byte	*
hlt	halt	
idiv	integer division	
idivb	integer division byte	*
imul	integer multiplication	
imulb	integer multiplication byte	*
in	input byte	*
inc	increment by one	
incb	increment byte by one	*

int	interrupt	
into	interrupt if overflow	
inw	input word	*
iret	interrupt return	
j	short jump	*
ja	short jump if above	
jae	short jump if above or equal	
jb	short jump if below	
jbe	short jump if below or equal	
jc	short jump if carry	
jcxz	short jump if CX is zero	
je	short jump on equal	
jg	short jump on greater than	
jge	short jump on greater than or equal	
jl	short jump on less than	
jle	short jump on less than or equal	
jmp	jump	
jmpi	inter segment jump	*
jna	short jump not above	
jnae	short jump not above or equal	
jnb	short jump not below	
jnbe	short jump not below or equal	
jnc	short jump not carry	
jne	short jump not equal	
jng	short jump not greater	
jnge	short jump not greater or equal	
jnl	short jump not less	
jnle	short jump not less or equal	
jno	short jump not overflow	
jnp	short jump not parity	
jns	short jump not sign	
jnz	short jump not zero	
jo	short jump on overflow	
jp	short jump if parity	
jpe	short jump if parity even	
jpo	short jump if parity odd	
js	short jump if signed	
jz	short jump if zero	
lahf	load AH from flags	
lds	load pointer using DS	
lea	load effective address	
les	load pointer using ES	
lock	lock bus	
lodb	load byte string	*
lodw	load word string	*
loop	loop short label	
loope	loop if equal	
loopne	loop if not equal	
loopnz	loop if not zero	
loopz	loop if zero	

mov	move	
movb	move byte	*
movs	move word string	*
movsb	move byte string	*
mul	multiplication unsigned	
mulb	multiplication unsigned byte	*
neg	negate	
negb	negate byte	*
nop	no operation	
not	logical NOT	
notb	logical NOT byte	*
or	logical OR	
orb	logical OR byte	*
out	output byte	*
outw	output word	*
pop	pop from stack	
popf	pop flag from stack	
push	push onto stack	
pushf	push flags onto stack	
rcl	rotate left through carry	
rclb	rotate left through carry byte	*
rcr	rotate right through carry	
rcrb	rotate right through carry byte	*
rep	repeat string op	*
repnz	repeat string op while not zero	*
repz	repeat string op while zero	*
ret	return from procedure	
reti	return from intersegment procedure	*
rol	rotate left	
rolb	rotate left byte	*
ror	rotate right	
rorb	rotate right byte	*
sahf	store AH into flags	
sal	shift arithmetic left	
salb	shift arithmetic left byte	*
sar	shift arithmetic right	
sarb	shift arithmetic right byte	*
sbb	subtract with borrow	
sbbb	subtract with borrow byte	*
scab	scan byte string	*
scaw	scan word string	*
shl	shift logical left	
shlb	shift logical left byte	*
shr	shift logical right	
shrb	shift logical right byte	*
stc	set carry flag	
std	set direction flag	
sti	set interrupt enable flag	
stob	store byte string	*
stow	store word string	*

sub	subtract	
subb	subtract byte	*
test	test	
testb	test byte	*
wait	wait while TEST pin	
xchg	exchange	
xchgb	exchange byte	*
xlat	translate	*
xor	exclusive OR	
xorb	exclusive OR byte	*

Byte Instructions

The XENIX assembler extends the definition of several instruction mnemonics to include an explicit byte "b" suffix. This suffix forces the operands in the instruction to be treated as bytes when they would otherwise be treated as words. There are the following byte instructions (not including byte string instructions):

adcb	imulb	rclb	shlb
addb	incb	rcrb	shrb
andb	movb	rolb	subb
cmpb	mulb	rorb	testb
decb	negb	salb	xchgb
divb	notb	sarb	xorb
idivb	orb	sbbb	

The byte instructions are especially useful when operating on memory operands defined with the **.byte** directive. Since **as** does not assign an explicit type to symbols created with the **.byte** or **.word** directives, it cannot detect the size of the associated item when given in an instruction. For example, if **test_byte** and **test_word** have been defined as

```
test_byte:      .byte   1
test_word:      .word   1
```

then the statements

```
negb        test_byte
neg         test_word
```

are required to operate on these values correctly. If **neg** were applied to **test_byte**, part of **test_word** would be destroyed in the operation.

Branch Instructions

The XENIX assembler has a new class of instructions, called branch instructions, that test a condition and branch to an instruction address further than 128 bytes away. These instructions take the same kind of operand as the normal **jmp** instruction but provide a test to see whether or not the jump should occur. The following is a list of the branch instructions.

beq	bhis	blt
bge	ble	bne
bgt	blo	br
bhi	blos	

The branch instructions, when assembled, consist of two 8086/286 jump instructions. The first jump tests for the inverse of the condition specified by the branch. The second jump is the unconditional jump instruction **jmp.** If the branch condition is true, the first jump is ignored and the second jump taken. If the branch condition is false, the first jump is taken. The first jump passes control to the next statement after the second jump.

For example, the statement

```
bne subtest
```

is equivalent to the statements

```
        je    no
        jmp   subtest
no:
```

String Instructions

The XENIX assembler uses a subset of the string instructions normally available for the 8086/286 processor. In particular, the assembler accepts only those string instructions that do not take operands. These are essentially the byte and word forms of the string instructions with implied destination.

To indicate this restriction, some instruction mnemonics no longer contain the "s" for string. The following is a list of the string instructions:

cmpsb	lodb	movsb	scab	stob
cmps	lodw	movs	scaw	stow

The assembler also accepts the **rep, repnz,** and **repz** instructions for repeating string operations. Note that these instructions must appear alone on a line; they cannot be combined on the same line with a string instruction.

Intersegment Instructions

The XENIX assembler has redefined call and jump instructions to create a new class of instructions called intersegment instructions. These allow calls and unconditional jumps to locations across 8086/286 segment boundaries. (These are physical segments, not the text, data, and bss segments described earlier.) There are the following intersegment instructions:

```
calli
jmpi
reti
```

The **calli** and **jmpi** instructions can have either a locally or globally defined symbol as an operand. In this case, an appropriate segment address is provided automatically when the program is linked. If an indirect address operand is used, an appropriate segment address must be explicitly provided. The **reti** instruction has operands similar to the **ret** instruction.

Input/Output Instructions

The XENIX assembler has modified the **in** and **out** instructions to include the new forms **inw** and **outw.** The **in** and **out** instructions now operate strictly on byte values; **inw** and **outw** operate on words. Furthermore, these instructions take only one operand--the port number. The **al** or **ax** register is accessed as appropriate and must not be given as an operand.

80286 Instructions

Assembly language programmers who wish to use 80286 instructions can insert the binary opcode of a given instruction into the instruction stream using the **.byte** and **.word** directives. For example, the directive

```
.byte  /6a,  *1
```

places the binary opcode of the **pushi** (for push immediate) instruction in the instruction stream. This code is equivalent to the 80286 instruction

```
pushi  *1
```

A programmer can also use the power of the C language preprocessor to create macros for the 80286 instructions. For example, if the macro definition

```
#define  PUSHI(x)    .byte /6a,  *x
```

is included in an assembly language source, then the macro call

```
PUSHI(1)
```

may be used for a **pushi** instruction in place of the **.byte** directive. In this case, you must invoke the C preprocessor using the **cc** command to resolve this macro.

Note that privileged 80286 instructions are not available to user programs.

Addressing Modes

The XENIX 8086/286 assembler provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory, or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways.

The following sections describe the format and meaning of instruction operands.

Register Operands

Register operands are the 8086/286 CPU registers. In an instruction, a register operand causes the instruction's action to be performed on the contents of the register. A register operand may be any one of the following:

ax	ah	al
bx	bh	bl
cx	ch	cl
dx	dh	dl
di	si	bp
sp	cs	ds
ss	es	

Register operands may be used for the source or destination in an instruction. Since these operands are encoded in a few bits, instructions that specify only register operands are generally the most compact. They are also the fastest, since operations on registers are performed entirely within the CPU.

The following are examples of instructions with register operands:

```
sub     ax,bx
addb    ah,dl
cmp     cs,ds
```

Immediate Operands

Immediate operands are byte or word constants given with the instruction itself. These operands have the forms

** expression*
 # expression

where ***** specifies a byte constant, **#** specifies a word constant, and *expression* is an absolute expression or a symbol that defines the constant's value.

Since immediate operands are constants, they cannot be used as the destination operand. Note that the assembler does not check the operand size.

The following examples illustrate immediate operands:

```
movb    cx,*33
mov     cx,#(122/2)
addb    ax,*NAME
```

Direct Address Operands

Direct address operands are the bytes or words in memory at the given direct addresses. Direct addresses have the form

expression

where *expression* is an absolute expression or symbol that resolves to a memory address. The direct address gives the location of the operand in terms of an offset from the beginning of the current segment or the segment in which the given symbol is defined.

Direct address operands may be used for source or destination. Although absolute addresses are allowed, symbols should be used whenever possible. The size of the operand depends on the instruction in which it is used.

The following examples illustrate direct address operands:

The following examples illustrate direct address operands:

```
mov     cx,free
movb    Darray + 204,*1
```

Based Operands

Based operands are bytes or words in memory with addresses computed by adding a constant and one of the base registers **bp** or **bx.** The operands have the forms

[*expression*] (bp)
[*expression*] (bx)

where *expression* is an absolute expression or symbol that resolves to an absolute.

Based operands are typically used to access structures. The base register points to the start of the structure, and items in the structure are addressed by an appropriate *expression.*

The following examples illustrate based operands:

```
mov     2(bp), #1000
movb    ax, TOP(bx)
neg     -4(bp)
mov     ax, (bx) (bp)
```

Indexed Operands

Indexed operands are bytes or words in memory with addresses computed by adding a constant and one of the index registers **di** or **si.** The operands have the forms

[*expression*] (di)
[*expression*] (si)

where *expression* is an absolute expression or symbol that resolves to an absolute.

Indexed operands are often used to access elements in an array. *expression* points to the start of the array. The index register is given the index value of the element to be accessed. Since all array elements are the same length, simple arithmetic on the index register will select any element.

The following examples illustrate indexed operands:

```
movb    ax,Darray(di)
addb    4096(si),*1
```

Based Indexed Operands

Based indexed operands are bytes or words in memory whose addresses are computed by adding a constant, a base register, and an index register. The operands have the forms

[*expression*] (bx) (di)
[*expression*] (bx) (si)
[*expression*] (bp) (di)
[*expression*] (bp) (si)

where *expression* is an absolute expression or a symbol that resolves to an absolute.

Based indexed operands provide a very flexible method of accessing items that require two address components. For example, elements of multidimensional arrays can be accessed by setting *expression* to the start address of the array and assigning the appropriately scaled index values to the base and index registers.

The following examples illustrate based indexed operands:

```
movb    Darray(bx)(di), *1
mov     ax, (bx)(si)
neg     -2(bp)(si)
```

Indirect Address Operands

Indirect address operands are instruction addresses stored in memory at given indirect addresses. Indirect address operands have the form

@expression

where *expression* is an absolute expression or a symbol that resolves to an absolute.

Indirect address operands may be used only with the **calli, call, jmpi,** and **jmp** instructions. When used with the intersegment call or jump instruction, *expression* must point to a 4-byte segment/offset instruction address. When used with the call or jump instruction, *expression* must point to a 2-byte offset to an instruction.

The following examples illustrate indirect address operands:

```
moncall:
        .word   OFFSET
        .word   SEG__SELECT
subtest:
        .word   OFFSET
        .text
examp:
        calli   @moncall
        jmp     @subtest
```

Diagnostics

When syntactic errors occur, the assembler displays the line number and the name of the file containing the error. If the errors are encountered in the first pass of the assembler, the second pass is canceled and no object file is created.

Error messages have the following form:

```
***ERROR*** syntax error, line nnn
file: eee errors
```

where *nnn* is the line number(s) containing the error, *file* is the name of the file, and *eee* is the total number of errors.

intel®

CHAPTER 8
csh: C SHELL

The C shell program, **csh**, is a command language interpreter for XENIX system users. The C shell, like the standard XENIX shell **sh**, is an interface between you and XENIX commands and programs. It translates command lines typed at a terminal into corresponding system actions, gives you access to information such as your login name, home directory, and mailbox, and supports the construction of shell procedures for automating system tasks.

This chapter explains how to use the C shell. It also explains the syntax and function of C shell commands and features and shows how to use these features to create shell procedures. The C shell is fully described in the entry **csh** in Appendix B, "Programming Commands."

Invoking the C Shell

You can invoke the C shell from another shell by typing the **csh** command

```
csh
```

at the standard shell's command line. You can also direct the system to invoke the C shell for you when you log in. If you have given the C shell as your login shell in your **/etc/passwd** file entry, the system automatically starts the C shell when you log in.

After the system starts the C shell, the shell searches your home directory for the command files **.cshrc** and **.login**. If the shell finds the files, it executes the commands contained in them and then displays the C shell prompt, normally a percent sign (%).

The **.cshrc** file typically contains commands to be executed each time you start a C shell, and the **.login** file contains the commands to be executed each time you log into the system. The following is an example of a typical **.login** file:

```
set ignoreeof
set mail = (/usr/spool/mail/bill)
set time = 15
set history = 10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C shell; there is no corresponding XENIX program for this command. In the example, **set** is used to set the C-shell variable **ignoreeof**, which shields the C shell from logging out if CONTROL-D is pressed. Instead of CONTROL-D, the **logout** command is used to log out of the system. Setting the **mail** variable in the example causes the C shell to notify you if you receive any mail in the specified mailbox.

The C shell variable **time** is set to 15, causing the C shell to automatically print statistics lines for commands that execute for at least 15 seconds of CPU time. The variable **history** is set to 10, indicating that the C shell will remember the last 10 commands typed in its history list (described later). Finally, the XENIX **mail** program is invoked.

When the C shell finishes processing the **.login** file, it begins reading commands from the terminal, prompting for each with

```
%
```

When you log out (by giving the **logout** command) the C shell prints

```
logout
```

and executes commands from the file **.logout** if it exists in your home directory. After that, the C shell terminates and XENIX logs you off the system.

Using Shell Variables

The C shell maintains a set of variables. For example, in the above discussion, the variables **history** and **time** had the values 10 and 15. Each C shell variable has as its value an array of zero or more strings. C shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is

```
set name = value
```

C shell variables can be used to store values to be used later in commands through a substitution mechanism. The C shell variables most commonly referenced are, however, those that the C shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C shell.

One of the most important variables is **path,** which contains a list of directory names. When you type a command name at your terminal, the C shell examines each named directory in turn until it finds an executable file with a name that corresponds to the name you typed. The **set** command with no arguments displays the values of all variables currently defined in the C shell. The following example shows typical default values:

```
argv        ()
home        /usr/bill
path        (. /bin /usr/bin)
prompt      %
shell  /bin/csh
status 0
```

This output indicates that the variable **path** begins with the current directory indicated by dot (.), then **/bin,** and last **/usr/bin.** Your own local commands may be in the current directory. Normal XENIX commands reside in **/bin** and **/usr/bin.**

Sometimes a number of locally developed programs reside in the directory **/usr/local.** If you want all C shells that you invoke to have access to these new programs, place the command

```
set path=(. /bin /usr/bin /usr/local)
```

in the **.cshrc** file in your home directory. Try doing this, then logging out and back in. Type

```
set
```

to see that the value assigned to **path** has changed.

When you log in, the C shell examines all directories in your path other than the current directory (.), to determine which commands are in those directories. The C shell remembers these commands in an internal table. This means that if a command is added to a directory in your search path after you have started the C shell, then the C shell may not find that command if you attempt to invoke it. If you want to use a command that has been added after you have logged in, give the command

```
rehash
```

to the C shell. **rehash** causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Note that rehashing is not necessary for commands added to the current directory. Since the C shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built-in variables are **home,** which shows your home directory, and **ignoreeof,** which can be set in your **.login** file to tell the C shell not to exit when it receives an end-of-file from a terminal. The variable **ignoreeof** is one of several variables with values that the C shell does not care about; the C shell is only concerned with whether these variables are set or unset. Thus, to set **ignoreeof** you simply type

```
set ignoreeof
```

and to unset it type

```
unset ignoreeof
```

Some other useful built-in C shell variables are **noclobber** and **mail.** The syntax

```
> filename
```

which redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a valuable file. If you prefer that the C shell not overwrite files in this way, type

```
set noclobber
```

in your **.login** file. After setting **noclobber,** typing

```
date > now
```

causes an error message if the file **now** already exists. You can type

```
date >! now
```

if you really want to overwrite the contents of **now.** The ">!" is a special syntax indicating that overwriting or "clobbering" the file is permitted. (The space between the exclamation point (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism (described below) and would have a very different effect.)

Using the C Shell History List

The C shell can maintain a history list containing the text of previous commands. You can use a notation that reuses commands, or words from commands, in forming new commands. This notation can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following example gives a sample session involving typical usage of the history mechanism of the C shell. Boldface indicates user input.

```
% cat bug.c
main( )

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4:  newline in string or char constant
"bug.c", line 5:  syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\n/p
        printf("hello\n");
w
32
```

```
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out:   2784 + 364 + 1028  =  4176b  =  0x1050b
bug:   2784 + 364 + 1028  =  4176b  =  0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x  1 bill  3932 Dec  19  09:41  a.out
-rwxr-xr-x  1 bill  3932 Dec  19  09:42  bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

In this example, we have a very simple C program that has a bug or two in the file **bug.c**, which we display using **cat**. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor that stands for the end-of-line. The C shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command that started with the letter "c". If there were other commands beginning with the letter "c" executed recently, we could have said "!cc" or even "!cc:p", which prints the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting **a.out** file, and then noting that there was still a bug, ran the editor again. After fixing the program, we ran the C compiler again, but added to the command an extra "-o bug" telling the compiler to place the resultant binary in the file **bug** rather than **a.out**. In general, the history mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created are, and then we ran an "ls -l" command with the same argument list, denoting the argument list:

```
!*
```

We then ran the program **bug** to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file **bug.c.** To print the listing, we piped the output to **lpr,** but misspelled it as "lpt". To correct this, we used a C shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with

```
!!
```

and sent its output to the printer.

Other mechanisms are available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. You can also refer to a previous command by searching for a string that appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the entry **csh** in Appendix B, "Programming Commands."

Using Aliases

The C shell has an alias mechanism that can be used to make transformations on commands immediately after they are input. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can also be obtained using C shell command files, but these take place in another instance of the C shell and cannot directly affect the current C shell's environment or involve commands such as **cd** which must be done in the current C shell.

For example, suppose there is a new version of the mail program on the system called **newmail** that you wish to use instead of the standard mail program **mail.** If you place the C shell command

```
alias mail newmail
```

in your **.cshrc** file, the C shell will transform an input line of the form

```
mail bill
```

into a call to **newmail.** Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **-s** option. You can accomplish this with the following **alias** command:

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command named **dir.** If we then type

```
dir ~bill
```

the C shell translates this to

```
ls -s /usr/bill
```

Note that the tilde (~) is a special C shell symbol that represents the user's home directory.

Thus the **alias** command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. You can also define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

specifies an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent most substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!*" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly, the following example defines a command that looks up its first argument in the password file.

```
alias whois 'grep \!^ /etc/passwd'
```

The C shell reads the **.cshrc** file each time it starts up. If you place a large number of aliases there, C shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15). Having too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

Redirecting Input and Output

In addition to the standard output, commands also have a standard error output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to redirect the standard error output along with the standard output. For instance, if you want to redirect the output of a long-running command into a file and wish to have a record of any error message it produces, you can type

command >& *file*

The ">&" here tells the C shell to route both the standard error output and the standard output into *file*. Similarly, you can give a command of the form

```
command |& lpr
```

to route both standard and standard error output through the pipe to the line printer.

The form

command >&! *file*

is used when **noclobber** is set and *file* already exists.

Finally, use the form

command >> *file*

to append output to the end of an existing file. If **noclobber** is set, then an error results if *file* does not exist, otherwise the C shell creates *file*. The form

command >>! *file*

appends to a file even if it does not exist and **noclobber** is set.

Creating Background and Foreground Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the C shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the C shell creates a job. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is typed at the end of a command pipeline or sequence, then the job is started as a background job. This means that the C shell does not wait for the job to finish, but immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C shell. Thus

```
du > usage &
```

runs the **du** program, which reports on the disk usage of your working directory, directs the output into the file **usage**, and returns immediately with a prompt for the next command without waiting for **du** to finish. The **du** program continues executing in the background until it finishes, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the INTERRUPT or QUIT signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

Using Built-In Commands

This section explains how to use some of the built-in C shell commands.

The **alias** command (see "Using Aliases" above) is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. **alias** can also be given one argument to show the current alias for a given string of characters. For example

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events difficult to reference contextually. There is also a C shell variable named **prompt.** By placing an exclamation point (!) in its value the C shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could type

```
set prompt='\! % '
```

Note that the exclamation mark (!) had to be escaped even within single quotes.

The **logout** command is used to terminate a login C shell that has **ignoreeof** set.

The **rehash** command causes the C shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C shell's search path and want the C shell to find it, since otherwise the hashing algorithm may tell the C shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. For example, to make five copies of the file **one** in the file **five** you could type

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

sets the value of the environment variable **TERM** to "adm3a". The **env** command prints out the environment. For example:

```
% env
HOME = /usr/bill
SHELL = /bin/csh
PATH = :/usr/ucb:/bin:/usr/bin:/usr/local
TERM = adm3a
USER = bill
```

The **source** command is used to cause the current C shell to read commands from a file until the end of the file. Thus

```
source .cshrc
```

can be used after editing in a change to the **.cshrc** file that you wish to take effect before the next time you log in.

The **time** command is used to cause a command to be timed no matter how much CPU time it takes. Thus

```
time cp /etc/rc /usr/bill/rc
```

displays

```
0.0u 0.1s 0:01 8%
```

Similarly

```
time wc /etc/rc /usr/bill/rc
```

displays

```
      52       178      1347   /etc/rc
      52       178      1347   /usr/bill/rc
      104      356      2694   total
0.1u 0.1s 0:00 13%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about one-tenth of a second system time (s); the elapsed time was one second (0:01). The word count command **wc** used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands are used to remove aliases and variable definitions from the C shell. The command **unsetenv** removes variables from the environment.

Creating Command Scripts

You can place commands in files and cause C shells to be invoked to read and execute commands from these files, which are called C shell scripts. This section describes the C shell features that are useful when creating C shell scripts.

Using the argv Variable

A **csh** command script can be interpreted by typing

```
csh script [argument]...
```

where *script* is the name of the file containing a group of C shell commands and *argument* is a sequence of command arguments. The C shell places these arguments in the variable **argv** and then begins to read commands from *script*. These parameters are then available through the same mechanisms used to reference any other C shell variables.

If you make the file **script** executable by typing

```
chmod 755 script
```

or

```
chmod +x script
```

and then place a C shell comment at the beginning of the C shell script (i.e., begin the file with a number sign (#)), then **/bin/csh** will automatically be invoked to execute **script** when you type

```
script
```

If the file does not begin with a number sign (#), then the C shell will call the standard shell **/bin/sh** to execute it. Thus from the C shell, you can execute scripts written for either the C shell or the standard shell. In the standard shell **sh**, you can only execute scripts written for **sh** and not scripts written for the C shell.

Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed, a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script, would cause the current value of the variable **argv** to be echoed to the output of the C shell script. It is an error for **argv** to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

You can also access the components of a variable that has several values. Thus

```
$argv[1]
```

gives the first component of **argv** or in the example above "a". Similarly

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give "a b". Other notations useful in C shell scripts are

```
$n
```

where *n* is an integer. This is shorthand for

```
$argv[ n ]
```

the *n*th parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current C shell. Since this process number is unique in the system, it is often used in the generation of unique temporary file names.

The form

```
$<
```

is quite special and is replaced by the next line of input read from the C shell's standard input (not the script it is reading). This is useful for writing interactive C shell scripts that read commands from the terminal, or for writing a C shell script that acts as a filter, reading lines from its input file. Thus, the sequence

```
echo -n 'yes or no?'
set a=($<)
```

writes out the prompt

```
yes or no?
```

without a newline and then reads the answer into the variable **a**. In this case **$#a** is 0 if either a blank line or CONTROL-D is typed.

One minor difference between **$***n* and **$argv[***n***]** should be noted here. The form **$argv[***n***]** will yield an error if *n* is not in the range 1-**$#argv** while **$***n* will never yield an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that it is never an error to give a subrange of the form *n*-; if there are less than *n* components of the given variable then no words are substituted. A range of the form *m*-*n* likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

Using Expressions

To construct useful C shell scripts, the C shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C shell with the same precedence that they have in C. In particular, the operations == and != compare strings, and the operators **&&** and **||** implement the logical AND and OR operations. The special operators =~ and !~ are similar to == and != except that the string on the right side can have pattern matching characters (like *, ?, or [and]). These operators test whether the string on the left matches the pattern on the right.

The C shell also supports file enquiries of the form

```
-option filename
```

where *option* is one of a number of single characters. For example, the expression primitive

```
-e filename
```

tells whether *filename* exists. Other primitives test for read, write, or execute access to the file, whether it is a directory, or if it has nonzero length.

You can test whether a command terminates normally, by using a primitive of the form

{ *command* }

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the **status** variable examined in the next command. Since **$status** is set by every command, its value is always changing.

For the full list of expression components, see the entry **csh** in Appendix B, "Programming Commands."

Using the C Shell: A Sample Script

A sample C shell script follows that uses the expression mechanism of the C shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue   # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script uses the **foreach** command. The command executes the other commands between the **foreach** and the matching **end** for each of the values given between parentheses with the named variable **i**, which is set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop, the iteration variable (**i** in this case) has the value at the last iteration.

The variable **noglob** is set to prevent file name expansion of the members of **argv.** This is a good idea, in general, if the arguments to a C shell script are file names that have already been expanded or if the arguments may contain file name expansion metacharacters. You can also quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords in this statement is not flexible due to the current implementation of the C shell. The following two formats are not acceptable to the C shell:

```
if (expression) # Won't work!
then
        command
        ...
endif
```

and

```
if (expression) then command endif # Won't work
```

The C shell does have another form of the **if** statement:

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve "|", "&", or ";" and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif,** for example:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in C shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a file name or **:e** to extract the extension. Thus if the variable **i** has the value **/mnt/foo.bar** then

```
echo $i $i:r :e
```

produces

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing ".bar" and the **:e** modifier leaves only the "bar". Other modifiers take off the last component of a path name leaving the head **:h** or all but the last component of a path name leaving the tail **:t.** These modifiers are fully described in the entry **csh** in Appendix B, "Programming Commands." You can also use the command substitution mechanism to perform modifications on strings and then re-enter the C shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. Note also that the current implementation of the C shell limits the number of colon modifiers on a "$" substitution to 1. Thus

```
% echo $i :h:t
```

produces

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign (#) lexically introduces a C shell comment in C shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C shell. This character can be quoted using ' or \ to place it in an argument word.

Using Other Control Structures

The C shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

  ...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see **csh** in Appendix B. C programmers should note that **breaksw** is used to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C shell supports a **goto** statement, with labels looking like they do in C, e.g.:

```
loop:
        commands
        goto loop
```

Supplying Input to Commands

Commands run from C shell scripts receive by default the standard input of the C shell running the script. This feature enables C shell scripts to fully participate in pipelines but mandates extra notation for commands that take inline data.

Thus we need a metanotation for supplying inline data to commands in C shell scripts. For example, consider this script that runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed -  << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation

```
<< 'EOF'
```

means that the standard input for the **ed** command is to come from the text in the C shell script file up to the next line consisting of exactly EOF. Because EOF is quoted in single quotation marks ('), the C shell does not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" that the C shell uses to terminate the text to be given to the command is quoted, then these substitutions are not performed. In the example, since we used the form "1,$" in our editor script, we needed to insure that the dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\), i.e.:

```
1, \$s/^[ ]*/ /
```

Quoting the EOF terminator is a more reliable way of achieving the same end.

Catching Interrupts

If our C shell script creates temporary files, we may wish to catch interruptions of the C shell script so that we can clean up these files. We can then do

```
onintr label
```

where **label** is a label in our program. If an interrupt is received, the C shell will do a "goto label" and we can remove the temporary files, then do an **exit** command (which is built in to the C shell) to exit from the C shell script. If we wish to exit with nonzero status, we can write

```
exit(1)
```

to exit with status 1.

Using Other Features

This section describes other features useful to writers of C shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the C shell. The **-n** option causes the C shell only to read commands and not to execute them and may sometimes be of use.

Remember that the C shell will not execute C shell scripts that do not begin with the number sign character (#), i.e., C shell scripts that do not begin with a comment.

The C shell provides another quotation mechanism, using double quotation marks ("). This other mechanism allows only some of the expansions discussed above, and serves to make the quoted string into a single word, just as single quotation marks (') do.

Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells on a particular system, **/bin/sh, /bin/nsh,** and **/bin/csh,** you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are very similar, we can use **foreach** to simplify them:

```
$ foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C shell prompts for input with "? " when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of file names or other words. For example, examine the following terminal session:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gave the variable **a** a list of all the file names in the current directory as its value. We can then iterate over these names to perform any chosen function.

When the C shell encounters a command enclosed in accent marks (`), it executes the command and takes the standard output of the command as the value of the expression formed by the delimited command.

The output of a command within accent marks (`) is converted by the C shell to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier **:x** exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

Using Braces with Arguments

Another form of file name expansion involves the characters { and }. These characters specify that the contained strings, separated by commas (,), are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

A{ *str1, str2,...str3* }B

expands to

A*str1*B A*str2*B ... A*strn*B

This expansion occurs before the other file name expansions and can be applied recursively (i.e., nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting file names are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments that are not file names but that have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories **hdrs, retrofit,** and **csh** in your home directory. This mechanism is most useful when the common prefix is longer, for example:

```
chown root /usr/demo/{file1,file2,...}
```

Substituting Commands

A command enclosed in accent marks (`) is replaced, just before file names are expanded, by the output from that command. Thus, you can type

```
set pwd = `pwd`
```

to save the current directory in the variable **pwd** or type

```
vi `grep -l TRACE *.c`
```

to run the editor **vi,** supplying as arguments those files whose names end in **.c** which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotation marks ("). Refer to **csh** in Appendix B for more information.

Special Characters

The following list summarizes the special characters recognized by **csh.**

Syntactic metacharacters

; Separates commands to be executed sequentially

| Separates commands in a pipeline

( ) Brackets expressions and variable values

& Follows commands to be executed without waiting for completion

File name metacharacters

/ Separates components of a file's path name

. Separates root parts of a file name from extensions

? Expansion character matching any single character

* Expansion character matching any sequence of characters

[ ] Expansion sequence matching any single character from a set of characters

~ Used at the beginning of a file name to indicate home directory

{ } Used to specify groups of arguments with common parts

Quotation metacharacters

\ Treat following single character as literal

' Treat enclosed characters as literal

` Like ', but allows variable and command expansion

Input/output metacharacters

< Indicates redirected input

> Indicates redirected output

Expansion/substitution metacharacters

$ Indicates variable substitution

! Indicates history substitution

: Precedes substitution modifiers

^ Used in special forms of history substitution

` Indicates command substitution

Other metacharacters

Begins scratch file names; indicates C shell comments

- Prefixes option (flag) arguments to commands

intel®

CHAPTER 9
lex: LEXICAL ANALYZER GENERATOR

lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching and produces a C program that recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to **lex**. The **lex** code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

The user supplies the additional code needed to complete his tasks, including code written by other generators. The program that recognizes the expressions is generated from the user's C program fragments. **lex** is not a complete language, but rather a generator representing a new language feature added on top of the C programming language.

lex turns the user's expressions and actions (called "source" in this chapter) into a C program named **yylex**. The **yylex** program recognizes expressions in a stream (called "input" in this chapter) and performs the specified actions for each expression as it is detected.

Consider a program to delete from the input all blanks or tabs at the ends of lines. The following source

```
%%
[ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules and one rule. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates one or more of the previous item; and the dollar sign ($) indicates the end of the line. No action is specified, so the program generated by **lex** will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$ ;
[ \t]+      printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observes at the termination of the string of blanks or tabs whether or not there is a newline character, and then executes the desired rule's action. The first rule matches all strings of blanks or tabs at the ends of lines, and the second rule matches all remaining strings of blanks or tabs.

lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. **lex** can also be used with a parser generator to perform the lexical analysis phase; it is especially easy to interface **lex** and **yacc.** **lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars but that require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex.** **yacc** users will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number of **lex** rules or the complexity of the rules is not important in determining speed, unless rules that include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex.**

In the program written by **lex,** the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

lex is not limited to source that can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for **ab** and another for **abcdefg,** and the input stream is **abcdefh, lex** will recognize **ab** and leave the input pointer just before **cd.** Such backup is more costly than the processing of simpler languages.

lex Source Format

The general format of **lex** source is

```
{ definitions }
%%
{ rules }
%%
{ user-subroutines }
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum **lex** program is thus

```
%%
```

(no definitions, no rules), which translates into a program that copies the input to the output unchanged.

In the **lex** program format shown above, the rules represent the user's control decisions. They make up a table in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus the following individual rule might appear:

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

This looks for the string **integer** in the input stream and prints the message

```
found keyword INT
```

whenever it appears in the input text. In this example the C library function **printf** is used to print the string. The end of the **lex** regular expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose you want to change a number of words from British to American spelling. **lex** rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word **petroleum** would become **gaseum;** a way of dealing with such problems is described in a later section.

lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (that match the corresponding characters in the strings being compared) and operator characters (that specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters. Thus, the regular expression

```
integer
```

matches the string **integer** wherever it appears, and the expression

```
a57D
```

looks for the string **a57D.**

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

If any of these characters are to be used literally, they must be quoted individually with a backslash (\) or as a group within quotation marks ("). Whatever is contained between a pair of quotation marks is to be taken as text characters. Thus

```
xyz" + + "
```

matches the string **xyz++** when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz + + "
```

is the same as the one above. Thus by quoting every nonalphanumeric character being used as a text character, you need not memorize the above list of current operator characters.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in

```
xyz\ + \ +
```

which is another, less readable, equivalent of the above expressions. The quoting mechanism can also be used to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within brackets must be quoted. Several normal C escapes with the backslash (\) are recognized:

\n	newline
\t	tab
\b	backspace
\\	backslash

Since newline is illegal in an expression, **\n** must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline, backspace, and backslash is always a text character.

Invoking lex

There are two steps in compiling a **lex** source program. First, the **lex** source must be turned into a generated C program. Then this program must be compiled and loaded, usually with a library of **lex** subroutines. The generated program is in a file named **lex.yy.c**. The **lex** I/O library is defined in terms of the C standard library.

The library is accessed by the linker flag **-ll.** So an appropriate set of commands is

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

The resulting program is placed in the usual file **a.out** for later execution. To use **lex** with **yacc** see the section "lex and yacc" later in this chapter and also Chapter 10, "yacc: Compiler-Compiler." Although the default **lex** I/O routines use the C standard library, the **lex** automata themselves do not do so. If private versions of **input, output,** and **unput** are given, the standard C library can be avoided.

Specifying Character Classes

Classes of characters can be specified using brackets: [and]. The construction

```
[abc]
```

matches a single character, which may be **a, b,** or **c.** Within square brackets, most operator meanings are ignored. Only three characters are special: the backslash (\), the hyphen (-), and the caret (^). The hyphen indicates ranges. For example

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underscore. Ranges may be given in either ascending or descending order. Using the hyphen between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and causes a warning message. If you want the hyphen in a character class, it should be first or last; thus

```
[- + 0-9]
```

matches all the digits and the plus and minus signs.

In character classes, the caret (^) operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except **a, b,** or **c,** including all special or control characters; or

```
[^a-zA-Z]
```

is any character that is not a letter. The backslash (\) provides an escape mechanism within character class brackets, so that characters can be entered literally by preceding them with this character. Escaping into octal is possible although nonportable. For example

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Specifying an Arbitrary Character

To match almost any character, the period (.) designates the class of all characters except a newline.

Specifying Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus

```
ab?c
```

matches either **ac** or **abc**. Note that the meaning of the question mark here differs from its meaning in the shell.

Specifying Repeated Expressions

Repetitions of classes are indicated by the asterisk (*) and plus (+) operators. For example

```
a*
```

matches any number of consecutive **a** characters, including zero, while **a+** matches one or more instances of **a**. For example

```
[a-z]+
```

matches all strings of lowercase letters, and

```
[A-Za-z][A-Za-z0-9]*
```

matches all alphanumeric strings with a leading alphabetic character; this is a typical expression for recognizing identifiers in computer languages.

Specifying Alternation and Grouping

The vertical bar (|) operator indicates alternation. For example

```
(ab | cd)
```

matches either **ab** or **cd.** Note that parentheses are used for grouping, although they are not necessary at the outside level. For example

```
ab | cd
```

would have sufficed in the preceding example. Parentheses should be used for more complex expressions, such as

```
(ab | cd + )?(ef)*
```

which matches such strings as **abefef, efefef, cdef,** and **cddd,** but not **abc, abcd,** or **abcdef.**

Specifying Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are the caret (^) and the dollar sign ($). If the first character of an expression is a caret, then the expression is only matched at the beginning of a line (after a newline character or at the beginning of the input stream). This can never conflict with the other meaning of the caret, complementation of character classes, since complementation applies only within brackets. If the very last character is dollar sign, the expression is only matched at the end of a line (when immediately followed by a newline). The latter operator is a special case of the slash (/) operator, which indicates trailing context. The expression

```
ab/cd
```

matches the string **ab,** but only if followed by **cd.** Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in **lex** by specifying start conditions as explained in the section "Specifying Left Context Sensitivity." If a rule is only to be executed when the **lex** automaton interpreter is in start condition **x,** the rule should be enclosed in angle brackets:

```
<x>
```

If we considered being at the beginning of a line to be start condition ONE, then the caret (^) operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Specifying Expression Repetition

Within the rules section of a lex source, curly braces () specify repetition. For example

```
a{1,5}
```

looks for 1 to 5 occurrences of the character **a.**

Specifying Definitions

Definitions are given in the first part of the **lex** source, before the rules. The curly braces ({ and }) specify definition expansion if they enclose a name. For example

```
{digit}
```

looks for a predefined string named **digit** and inserts it at that point in the expression.

Specifying Actions

When an expression is matched by a pattern of text in the input, **lex** executes the corresponding action. This section describes some features of **lex** that aid in writing actions. Note that there is a default action, copying the input to the output. This is performed on all strings not otherwise matched. Thus the **lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **lex** is being used with **yacc,** this is the normal situation. You may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule that merely copies can be omitted.

One of the simplest actions is to ignore the input. Specifying a C null statement **;** as an action causes this result. A frequent rule is

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

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is to use the repeat action character, |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in a different style. The quotes around **\n** and **\t** are not required.

In more complex actions, you often want to know the actual text that matched some expression like

```
[a-z] +
```

lex leaves this text in an external character array named **yytext.** Thus, to print the name found, a rule like

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

prints the string in **yytext.** The C function **printf** accepts a format argument and data to be printed; in this case, the format is **print string** where the percent sign (%) indicates data conversion, **s** indicates string type, and the data are the characters in **yytext.** So this just places the matched string on the output. This action is so common that it can be written as **ECHO.** For example

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

is the same as the preceding example. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule that is not desired. For example, if a rule matches **read,** it will normally match the instances of **read** contained in **bread** or **readjust;** to avoid this, a rule of the form

```
[a-z] +
```

is needed. This is explained further below, in the section "Handling Ambiguous Source Rules."

Sometimes it is more convenient to know the end of what has been found; hence **lex** also provides a count of the number of characters matched in the variable **yyleng.** To count both the number of words and the number of characters in words in the input, you might write

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

which accumulates in the variable **chars** the number of characters in the words recognized. The last character in the string matched can be accessed with

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

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, **yymore** can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string will overwrite the current entry in **yytext.** Second, **yyless(***n***)** may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in **yytext** to be retained. Further characters previously matched are returned to the input. This provides the same sort of look-ahead offered by the slash (/) operator, but in a different form.

For example, consider a language that defines a string as a set of characters between quotation marks (") and provides that to include a quotation mark in a string, it must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so that it might be preferable to write

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

which when faced with a string such as

```
"abc\"def"
```

will first match the five characters

```
"abc\
```

and then the call to **yymore** will cause the next part of the string

```
"def
```

to be tacked on the end. Note that the final quotation mark terminating the string should be picked up in the code labeled normal processing.

The function **yyless** might be used to reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =-*a*. Suppose you want to treat this as =- *a* and to print a message. A rule might be

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

which prints a message, returns the letter after the operator to the input stream, and treats the operator as =-.

Or you might want to treat this as = -*a*. To do this, just return the minus sign as well as the letter to the input. The following performs the interpretation:

```
=-[a-zA-Z]    {
        printf("Operator (=-)  ambiguous\n");
        yyless(yyleng-2);
        ... action for  = ...
        }
```

Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second: no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =-**3** however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, **lex** also permits access to the I/O routines it uses. They include

- **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**.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by **input** must mean end-of-file; and the relationship between **unput** and **input** must be retained or the look-ahead will not work. **lex** does not look ahead at all if it does not have to, but every rule containing a slash (/) or ending in one of the following characters implies look-ahead:

```
+ * ? $
```

Look-ahead is also necessary to match an expression that is a prefix of another expression. The standard **lex** library imposes a 100-character limit on backup.

Another **lex** library routine that you sometimes want to redefine is **yywrap,** which is called whenever **lex** reaches an end-of-file. If **yywrap** returns a 1, **lex** continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a **yywrap** that arranges for new input and returns 0. This instructs **lex** to continue processing. The default **yywrap** always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that you cannot write a normal rule that recognizes end-of-file; the only access to this condition is through **yywrap.** In fact, unless a private version of **input** is supplied, a file containing nulls cannot be handled, since a value of 0 returned by **input** is taken to be end-of-file.

Handling Ambiguous Source Rules

lex can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses as follows:

- The longest match is preferred.
- Among rules that match the same number of characters, the first given rule is preferred.

For example, suppose the following rules are given:

```
integer     keyword action ...;
[a-z]+      identifier action ...;
```

If the input is **integers,** it is taken as an identifier, because

```
[a-z]+
```

matches eight characters while

```
integer
```

matches only seven. If the input is **integer,** both rules match seven characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., **int**) does not match the expression **integer,** so the identifier interpretation is used.

The principle of preferring the longest match makes certain constructions dangerous, such as the following:

```
.*
```

For example

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

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

the above expression matches

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

which is probably not what was wanted. A better rule is of the form

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

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match a newline. Therefore, no more than one line is ever matched by such expressions. Don't try to defeat this with expressions like

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

or their equivalents; the **lex** generated program will try to read the entire input file, causing internal buffer overflows.

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for only once. For example, suppose you want to count occurrences of both **she** and **he** in an input text. Some **lex** rules to do this might be

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

where the last two rules ignore everything besides **he** and **she.** Remember that the period (.) does not include the newline. Since **she** includes **he, lex** will normally not recognize the instances of **he** included in **she,** since once it has passed a **she** those characters are gone.

Sometimes the user would like to override this choice. The action **REJECT** means go do the next alternative. It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose you really want to count the included instances of **he:**

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

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, you could note that **she** includes **he**, but not vice versa, and omit the **REJECT** action on **he**; in other cases, however, it would not be possible to tell which input characters were in both classes.

Consider the two rules

```
a[bc] +     { ... ; REJECT; }
a[cd] +     { ... ; REJECT; }
```

If the input is **ab**, only the first rule matches, and on **ad** only the second matches. The input string **accb** matches the first rule for four characters and then the second rule for three characters. In contrast, the input **accd** agrees with the second rule for four characters and then the first rule for three.

In general, **REJECT** is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap; e.g., the word **the** is considered to contain both **th** and **he**. Assuming a two-dimensional array named **digram** to be incremented, the appropriate source is

```
%%
[a-z][a-z]  { digram [ yytext [ 0 ] ] [ yytext [ 1 ] ] + + ; REJECT; }
.       ;
\n      ;
```

where the **REJECT** is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Remember that **REJECT** does not rescan the input. Instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and **REJECT** executed, you must not have used **unput** to change the characters forthcoming from the input stream. This is the only restriction on the ability to manipulate the not-yet-processed input.

Specifying Left Context Sensitivity

Sometimes you may want to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^) operator, for example, is a prior context operator, recognizing immediately preceding left context just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments:

- The use of flags, when only a few rules change from one environment to another
- The use of start conditions with rules
- The use of multiple lexical analyzers running together

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and that set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all. It may be more convenient, however, to have **lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word **magic** to **first** on every line that began with the letter **a**, changing **magic** to **second** on every line that began with the letter **b**, and changing **magic** to **third** on every line that began with the letter **c**. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      { flag = 'a'; ECHO; }
^b      { flag = 'b'; ECHO; }
^c      { flag = 'c'; ECHO; }
\n      { flag =  0 ; ECHO; }
magic
        {
        switch (flag)
        {
        case 'a':  printf("first"); break;
        case 'b':  printf("second"); break;
        case 'c':  printf("third"); break;
        default:  ECHO; break;
        }
        }
```

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word **Start** may be abbreviated to **s** or **S.** The conditions may be referenced at the head of a rule with angle brackets. For example

```
<name1> expression
```

is a rule that is only recognized when **lex** is in the start condition *name1.* To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1.* To return to the initial state

```
BEGIN 0;
```

resets the initial condition of the **lex** automaton interpreter. A rule may be active in several start conditions; for example

```
<name1, name2, name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written as:

```
%START AA BB CC
%%
^a          { ECHO; BEGIN AA; }
^b          { ECHO; BEGIN BB; }
^c          { ECHO; BEGIN CC; }
\n          { ECHO; BEGIN 0; }
<AA> magic      printf("first");
<BB> magic      printf("second");
<CC> magic      printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than the user's code.

Specifying Source Definitions

Remember the format of the **lex** source:

{ *definitions* }
%%
{ *rules* }
%%
{ *user-routines* }

So far only the rules have been described. You will need additional options, though, to define variables for use in your program and for use by **lex.** These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source not intercepted by **lex** is copied into the generated program. There are three types of copied source lines:

- Any line not part of a **lex** rule or action that begins with a blank or tab is copied into the **lex** generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **lex** that contains the actions. This material must look like program fragments and should precede the first **lex** rule.

  As a side effect of the above, lines that begin with a blank or tab, and that contain a comment, are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow the conventions of the C language.

- Anything included between lines containing only **%{** and **%}** is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1 or copying lines that do not look like programs.

- Anything after the third **%%** delimiter, regardless of formats, is copied out after the **lex** output.

Definitions intended for **lex** are given before the first **%%** delimiter. Any line in this section not contained between **%{** and **%}**, and beginning in column 1, is assumed to define **lex** substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {*name*} syntax in a rule. Using **{D}** for the digits and **{E}** for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                                   [0-9]
E                                [DEde][-+]?{D}+
%%
{D}+                        printf("integer");
{D}+"."{D}*({E})?           |
{D}*"."{D}+({E})?           |
{D}+{E}                     printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point, and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as **35.EQ.I,** which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ  printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. These possibilities are discussed later in the section "Source Format."

lex and yacc

If you want to use **lex** with **yacc,** note that what **lex** writes is a program named **yylex,** the name required by **yacc** for its analyzer. Normally, the default main program in the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** will call **yylex.** In this case, each **lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to access **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line

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

in the last section of **yacc** input. Suppose the grammar is to be named **good** and the lexical rules are to be named **better;** the XENIX command sequence can just be

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

The **yacc** library (**-ly**) should be loaded before the **lex** library to obtain a main program that invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable **lex** source program to do just that:

```
%%
                int k;
[0-9]+          {
                k = atoi(yytext);
                if (k%7 == 0)
                        printf("%d", k+3);
                else
                        printf("%d",k);
                }
```

The rule [0-9]+ recognizes strings of digits; **atoi** converts the digits to binary and stores the result in **k**. The remainder operator (%) is used to check whether **k** is divisible by 7; if it is, it is incremented by 3 as it is written out. Note that this program will alter such input items as 49.63 or X7 and will also increment the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                int k;
-?[0-9]+        {
                k = atoi(yytext);
                printf("%d", k%7 == 0 ? k+3 : k);
                }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a decimal point or preceded by a letter will be picked up by one of the last two rules and not changed. The **if-else** has been replaced by a C conditional expression to save space; the form **a?b:c** means "if **a** then **b** else **c**".

For an example of statistics gathering, here is a program that makes histograms of word lengths, where a word is defined as a string of letters:

```
                int lengs[100];
%%
[a-z]+          lengs[yyleng]++;
.               |
\n              ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
        if (lengs[i] > 0)
                printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement **return**(1); indicates that **lex** is to perform wrapup. If **yywrap** returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a **yywrap** that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written to convert double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish between upper- and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
.       .
.       .
.       .
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes **double precision** to **real**, or **DOUBLE PRECISION** to **REAL**:

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}{
        printf(yytext[0] = = 'd'? "real" : "REAL");
        }
```

Care is taken throughout this program to preserve the case of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"      "[^0]       ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as beginning of line, then five blanks, then anything but blank or zero. Note the two different meanings of the caret (^) here. Some rules then follow to change double precision constants to ordinary floating constants:

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
        /* convert constants */
        for(p = yytext; *p != 0; p++)
                {
                if (*p = = 'd' || *p = = 'D')
                        *p += 'e'-'d';
                ECHO;
                }
```

After the floating-point constant is recognized, it is scanned by the **for** loop to find the letter "d" or "D". The program then adds 'e'-'d', which converts it to the next letter of the alphabet. The modified constant, now single precision, is written out again. A series of names follows that must be respelled to remove their initial **d.** By using the array **yytext,** the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}        |
{d}{c}{o}{s}        |
{d}{s}{q}{r}{t}     |
{d}{a}{t}{a}{n}     |
...
{d}{f}{l}{o}{a}{t}      printf("%s",yytext + 1);
```

Another list of names must have initial **d** changed to initial **a:**

```
{d}{l}{o}{g}        |
{d}{l}{o}{g} 10     |
{d}{m }{i}{n} 1     |
{d}{m}{a}{x} 1      {
        yytext[0] + = 'a' -'d';
        ECHO;
        }
```

And one routine must have initial **d** changed to initial **r:**

```
{d} 1 {m}{a}{c}{h}   {
        yytext[0] + = 'r' - 'd';
        ECHO;
}
```

To avoid such names as **dsinx** being detected as instances of **dsin,** some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*        |
[0-9] +                     |
\n                          |
.           ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

Specifying Character Sets

The programs generated by **lex** handle character I/O only through the routines **input, output,** and **unput.** Thus the character representation provided in these routines is accepted by **lex** and employed to return values in **yytext.** For internal use, a character is represented as a small integer that, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter **a** is represented as the same form as the character constant:

```
'a'
```

If this interpretation is changed, by providing I/O routines that translate the characters, **lex** must be told about it, by giving a translation table. This table must be in the definitions section and must be bracketed by lines containing only **%T.** The table contains lines of the form

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

which indicate the value associated with each character. For example:

```
%T
1       Aa
2       Bb
...
26      Zz
27      \n
28       +
29      -
30      0
31      1
...
39      9
%T
```

This table maps the lowercase and uppercase letters together into the integers 1 through 26, newline into 27, plus (+) and minus (-) into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a larger number than is supported by the hardware character set.

Source Format

The general form of a **lex** source file is

{ *definitions* }
%%
{ *rules* }
%%
{ *user-subroutines* }

The definitions section contains a combination of

- Definitions, in the form "*name* space *translation*"
- Included code, in the form "space *code*"
- Included code, in the form

  %{
  code
  %}

- Start conditions, in the form

  %S *name1 name2* ...

- Character set tables, in the form

  %T
  number space *character-string*
  %T

- Changes to internal array sizes, in the form

  %*x nnn*

  where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form

expression action

where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

x	The character x
"x"	An x, even if x is an operator
\x	An x, even if x is an operator
[xy]	The character x or y
[x-z]	The characters x, y, or z
[^x]	Any character but x
.	Any character but newline
^x	An x at the beginning of a line
<y> x	An x when **lex** is in start condition y
x$	An x at the end of a line
x?	An optional x
x*	0,1,2, ... instances of x
x+	1,2,3, ... instances of x
x|y	An x or a y
(x)	An x
x/y	An x but only if followed by y
{xx}	The translation of xx from the definitions section
x{*m*,*n*}	*m* through *n* occurrences of x

intel®

CHAPTER 10
yacc: COMPILER-COMPILER

Computer program input generally has some structure; every computer program that does input can be thought of as defining an input language that it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

yacc provides a general tool for describing the input to a computer program. The name **yacc** itself stands for "yet another compiler-compiler." The **yacc** user specifies the structures of his or her input, together with code to be invoked as each such structure is recognized. **yacc** turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by **yacc** calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. The class of specifications accepted is a very general one: LALR grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., **yacc** has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. **yacc** then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (called the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

yacc is written in a portable dialect of C, and the actions and output subroutine are in C as well. Moreover, many of the syntactic conventions of **yacc** follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ', ' year   ;
```

Here, **date, month_name, day,** and **year** represent structures of interest in the input process; presumably, **month_name, day,** and **year** are defined elsewhere. The comma (,) is enclosed in single quotation marks; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
      .
      .
      .
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and **month_name** would be a nonterminal symbol. Such low-level rules tend to waste time and space and may complicate the specification beyond **yacc**'s ability to deal with it. Usually, the lexical analyzer would recognize the month names and return an indication that a **month_name** was seen; in this case, **month_name** would be a token.

Literal characters, such as the comma, must also be passed through the lexical analyzer and are considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be slipped in to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the re-entry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, constructions difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe

- The preparation of grammar rules
- The preparation of the user-supplied actions associated with the grammar rules
- The preparation of lexical analyzers
- The operation of the parser
- Various reasons why **yacc** may be unable to produce a parser from a specification, and what to do about it
- A simple mechanism for handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions that should improve the style and efficiency of the specifications

Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent (**%%**) marks. (The percent sign (%) is generally used in **yacc** specifications as an escape character.) In other words, a full specification file looks like this:

declarations
%%
rules
%%
programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal **yacc** specification is

%%
rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form

A : *BODY* ;

where *A* represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length and may be made up of letters, dot (.), the underscore (_), and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotation marks ('). As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized:

'\n'	=	Newline
'\r'	=	Return
'\''	=	Single quotation mark
'\\'	=	Backslash
'\t'	=	Tab
'\b'	=	Backspace
'\f'	=	Formfeed
'*xxx*'		A character value specified by the octal number *xxx*

For a number of technical reasons, the ASCII NULL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, then the vertical bar (|) can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

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

can be given to **yacc** as

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

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name 1 name2 ...
```

in the declarations section. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the **%start** keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token called the endmarker. If the tokens up to, but not including, the endmarker form a structure that matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate. Usually the endmarker represents some reasonably obvious I/O status, such as the end of the file or end of the record.

Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ({ and }). For example

```
A : '(' B ')'
      {   hello( 1, "abc" ); }
```

and

```
XXX : YYY ZZZ
      {     printf("a message\n");
            flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign ($) is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable **$$** to some value. For example, an action that does nothing but return the value 1 is

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

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ..., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

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

for example, then $2 has the value returned by C and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  : '(' expr ')' ;
```

The value returned by this rule is usually the value of the **expr** in parentheses. This can be indicated by

```
expr  : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to gain control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

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

the effect is to set **x** to 1 and **y** to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. **yacc** actually treats the above example as if it had been written as

```
$ACT : /* empty */
            { $$ = 1; }
      ;

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

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose a C function **node** is written so that the call

```
node( L, n1, n2 )
```

creates a node with label *L*, and descendants *n1* and *n2*, and returns the index of the newly created node. Then a parse tree can be built by supplying actions such as

```
expr : expr '+' expr
      { $$ = node('+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks **%{** and **%}**. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example

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

could be placed in the declarations section, making **variable** accessible to all of the actions. The **yacc** parser uses only names beginning in **yy;** the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in a later section.

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex.** The function returns an integer, called the token number, representing the kind of token read. If a value is associated with that token, it should be assigned to the external variable **yylval.**

The parser and the lexical analyzer must agree on these token numbers for communication between them to take place. The numbers may be chosen by **yacc** or by the user. In either case, the **#define** mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name **DIGIT** has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like this

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
        ...
        case '9':
                yylval = c - '0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of **DIGIT** and a value equal to the numeric value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier **DIGIT** will be defined as the token number associated with the token **DIGIT.**

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

As mentioned above, the token numbers may be chosen by **yacc** or by the user. In the default situation, the numbers are chosen by **yacc.** The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. All token numbers must be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is **lex,** discussed in Chapter 9. These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) that do not fit any theoretical framework, and for which lexical analyzers must be crafted by hand.

How the Parser Works

yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, however, is relatively simple, and understanding how it works will make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no look-ahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

- Based on its current state, the parser decides whether it needs a look-ahead token to decide what action should be done; if it needs one and does not have one, it calls **yylex** to obtain the next token.

- Using the current state, and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the look-ahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

```
IF      shift 34
```

which says if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule, replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to grammar rule 18, while the action

```
IF      shift 34
```

refers to state 34.

Suppose the rule being reduced is

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

The reduce action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack (in general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing **x**, **y**, and **z** and no longer serve any useful purpose. After popping these states, a state is uncovered that was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the look-ahead token is cleared by a shift and is not affected by a goto. In any case, the uncovered state contains an entry such as

```
A    goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the reduce action turns back the clock in the parse, popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable **yylval** is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable **yyval** is copied onto the value stack. The pseudo- variables **$1**, **$2**, etc. refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the endmarker and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the 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: Error recovery is described in the section "Error Handling," later in this chapter.

Consider the following example:

```
%token DING DONG DELL
%%
rhyme : sound place
      ;
sound : DING DONG
      ;
place : DELL
      ;
```

When **yacc** is invoked with the **-v** option, a file called **y.output** is produced, with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) is

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_  (1)

        . reduce 1

state 5
        place : DELL_  (3)

        . reduce 3

state 6
        sound : DING DONG_  (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The underscore character (_) is used to indicate what has been seen and what is yet to come in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input to decide between the actions available in state 0, so the first token, **DING**, is read, becoming the look-ahead token. The action in state 0 on **DING** is **shift 3**, so state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, **DONG**, is read, becoming the look-ahead token. The action in state 3 on the token **DONG** is **shift 6**, so state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right-hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on **sound**

```
sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, **DELL**, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on **place**, the left side of rule 3, is state 4. Now the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on **rhyme** causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by **$end** in the **y.output** file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

You should consider how the parser works when confronted with such incorrect strings as **DING DONG DONG, DING DONG, DING DONG DELL DELL,** etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

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

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

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

or as

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

(The first is called left association, the second right association).

yacc detects such ambiguities when attempting to build the parser. Consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to **expr** and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce/reduce conflict. Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat   : IF '(' cond ')' stat
       | IF '(' cond ')' stat ELSE stat
       ;
```

In these rules, **IF** and **ELSE** are tokens, **cond** is a nonterminal symbol describing conditional (logical) expressions, and **stat** is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each **ELSE** is associated with the last **IF** immediately preceding the **ELSE.** In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the **ELSE.** It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the **ELSE** may be shifted, **S2** read, and then the right-hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things; there is a shift/reduce conflict. The application of disambiguating Rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, **ELSE**, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift/reduce conflict  (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_    (18)
        stat : IF ( cond ) stat_ELSE stat

        ELSE  shift 45
        .   reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules that has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is **ELSE**, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the **ELSE** will have been shifted in this state. Back in state 23, the alternative action, described by "." , is to be done if the input symbol is not mentioned explicitly in the above actions; thus in this case, if the input symbol is not **ELSE**, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the **y.output** file, the rule numbers are printed after those rules that can be reduced. In most states, there will be at most one reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword (**%left, %right,** or **%nonassoc),** followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus

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

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator .LT. in FORTRAN, that may not associate with themselves; thus

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword **%nonassoc** in **yacc.** As an example of the behavior of these declarations, the description

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

%%

expr  : expr '=' expr
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | NAME
      ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword **%prec** changes the precedence level associated with a particular grammar rule. The **%prec** keyword appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

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

%%

expr  : expr'+'expr
      | expr'-'expr
      | expr'*'expr
      | expr'/'expr
      | '-'expr %prec'*'
      | NAME
      ;
```

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

- The precedences and associativities are recorded for those tokens and literals that have them.

- A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

- When there is a reduce/reduce conflict or a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

- If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences until some experience has been gained. The **y.output** file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to perform this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides a simple but reasonably general feature. The token name **error** is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token, and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

To prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Somewhat easier are rules such as

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

Here, when there is an error, the parser attempts to skip over the statement but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any cleanup action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be re-entered after an error. A possible error rule might be

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

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the re-entered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

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

As mentioned above, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by **yylex** would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error
            {     resynch( );
                  yyerrok ;
                  yyclearin ; }
            ;
```

These mechanisms are admittedly crude. They do allow for a simple, effective recovery of the parser from many errors. Moreover, the user can gain control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to **yacc,** the output is a file of C programs called **y.tab.c** on most systems. The function produced by **yacc** is called **yyparse;** it is an integer valued function. When it is called, it in turn repeatedly calls **yylex,** the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) **yyparse** returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, **yyparse** returns the value 0.

The user must provide a certain amount of environment for this parser to obtain a working program. For example, as with every C program, a program called **main** must be defined that eventually calls **yyparse.** In addition, a routine called **yyerror** prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc,** a library has been provided with default versions of **main** and **yyerror.** This library can be accessed with a **-ly** option to the linker. To show the triviality of these default programs, the source is given below:

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

and

```
#include <stdio.h>

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

The argument to **yyerror** is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable **yychar** contains the look-ahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the **main** program is probably supplied by the user (to read arguments, etc.) the **yacc** library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. **yydebug** can be set at run-time using a debugger such as **adb.**

Preparing Specifications

This section contains miscellaneous hints on preparing clear, efficient, and easy-to-change specifications.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. These hints will help:

- Use uppercase letters for token names and lowercase letters for nonterminal names. This rule helps you to know who to blame when things go wrong.
- Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
- Put all rules with the same left-hand side together. Put the left-hand side in only once, and let all following rules begin with a vertical bar.
- Put a semicolon only after the last rule with a given left-hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
- Indent rule bodies by two tab stops and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so-called left recursive grammar rules, rules of the form

name : *name rest-of-rule* ;

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |list ',' item
        ;
```

and

```
seq     :       item
        |seq item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq    :      item
       | item seq
       ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

Consider whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq    :      /* empty */
       | seq item
       ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen, when it hasn't seen enough to know.

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements.

Consider the following:

```
%{
int dflag;
%}
... other declarations ...

%%

prog   : decls stats
       ;

decls  : /* empty */
                    {   dflag = 1; }
       | decls declaration
       ;

stats  : /* empty */
                    {   dflag = 0; }
       | stats statement
       ;

... other rules ...
```

The flag **dflag** is now 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of approach can be overdone. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

Handling Reserved Words

Some programming languages permit the user to use words like **if**, that are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable." It is best that keywords be reserved, that is, be forbidden for use as variable names.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of the macros **YYACCEPT** and **YYERROR. YYACCEPT** causes **yyparse** to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; **yyerror** is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    : adj noun verb adj noun
              { look at the sentence ... }
        ;

adj     : THE { $$ = THE; }
        | YOUNG { $$ = YOUNG; }
        ...
        ;

noun    : DOG  { $$ = DOG; }
        | CRONE { if( $0 = = YOUNG ){
                      printf( "what?\n" );
                      }
                $$ = CRONE;
          }
        ;
        ...
```

In the action following the word **CRONE, a** check is made that the preceding token shifted was not **YOUNG.** Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Supporting Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser will be strictly type checked. The **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a **$$** or **$** construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as **lint** will be far more silent.

Three mechanisms are used for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the **yacc** value stack, and the external variables **yylval** and **yyval**, to have type equal to this union. If **yacc** was invoked with the **-d** option, the union declaration is copied onto the **y.tab.h** file. Alternatively, the union may be declared in a header file, and a **typedef** used to define the variable **YYSTYPE** to represent this union. Thus, the header file might also have said:

```
typedef union {
        body o f union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of **%{** and **%}**.

Once **YYSTYPE** is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no predefined type. Similarly, reference to left context values (such as **$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule : aaa { $<intval>$ = 3; } bbb
        { fun( $<intval>2, $<other>0 ); }
  ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in a later section. The facilities in this subsection are not triggered until they are used: in particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $ or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold **int**'s, as was true historically.

A Small Desk Calculator

This example gives the complete **yacc** specification for a small desk calculator: the desk calculator has 26 registers, labeled **a** through **z**, and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; this job is probably better done by the lexical analyzer.

```
/* desk calculator parser specification */
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

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

%%   /* beginning of rules section */

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

```
stat : expr
        { printf( "%d\n", $1 ); }
       | LETTER '=' expr
             { regs[$1] = $3; }
       ;

expr : '(' expr ')'
             { $$ = $2; }
       | expr '+' expr
             { $$ = $1 + $3; }
       | expr '-' expr
             { $$ = $1 - $3; }
       | expr '*' expr
             { $$ = $1 * $3; }
       | expr '/' expr
             { $$ = $1 /$3; }
       | expr '%' expr
             { $$ = $1 % $3; }
       | expr '&' expr
             { $$ = $1 & $3; }
       | expr '|' expr
             { $$ = $1 | $3; }
       | '-' expr %prec UMINUS
             { $$ = -$2; }
       | LETTER
             { $$ = regs[$1]; }
       | number
       ;

number : DIGIT
             { $$ = $1; base = ($1 = = 0) ? 8 : 10; }
       | number DIGIT
             { $$ = base * $1 + $2; }
       ;

%%    /* start of programs */

yylex() {   /* lexical analysis routine */
              /* returns LETTER for a lowercase letter, */
              /* yylval = 0 through 25 */
              /* return DIGIT for a digit, */
              /* yylval = 0 through 9 */
              /* all other characters */
              /* are returned immediately */

       int c;

       while( (c = getchar( ) ) = = ' ' ) { /* skip blanks */ }

       /* c is now nonblank */
```

```
        if( islower( c ) ) {
                yylval = c-'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c-'0';
                return( DIGIT );
                }
        return( c );
        }
```

yacc Input Syntax

This section has a description of the **yacc** input syntax, as a **yacc** specification. Context dependencies are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the difficult part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token **C_IDENTIFIER.** Otherwise, it returns **IDENTIFIER.** Literals (quoted strings) are also returned as **IDENTIFIER,** but never as part of **C_IDENTIFIER.**

```
                /* grammar for the input to yacc */

        /* basic entities */
%token IDENTIFIER    /* includes identifiers and literals */
%token C_IDENTIFIER  /* identifier followed by colon     */
%token NUMBER/* [0-9] + */

        /* reserved words: %type = > TYPE, %left = > LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK  /* the %% mark */
%token LCURL /* the % { mark */
%token RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start spec

%%

spec  : defs MARK rules tail
        ;

tail  : MARK  { Eat up the rest of the file }
        | /* empty: the second MARK is optional */
        ;
```

```
defs      : /* empty */
          |defs def
          ;

def    : START IDENTIFIER
          |UNION { Copy union definition to output }
          |LCURL { Copy C code to output file } RCURL
          |ndefs rword tag nlist
          ;

rword  : TOKEN
          |LEFT
          |RIGHT
          |NONASSOC
          |TYPE
          ;

tag    : /* empty: union tag is optional */
          |'<' IDENTIFIER '>'
          ;

nlist  : nmno
          |nlist nmno
          |nlist ',' nmno
          ;

nmno   : IDENTIFIER      /* Literal illegal with %type */
          |IDENTIFIER NUMBER /* Illegal with %type */
          ;

          /* rules section */

rules  : C_IDENTIFIER rbody prec
          |rules rule
          ;

rule   : C_IDENTIFIER rbody prec
          |'|' rbody prec
          ;

rbody  : /* empty */
          |rbody IDENTIFIER
          |rbody act
          ;

act    : '{' { Copy action, translate $$, etc. } '}'
          ;

prec   : /* empty */
          |PREC IDENTIFIER
          |PREC IDENTIFIER act
          |prec ';'
          ;
```

An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating-point interval arithmetic. The calculator understands floating-point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating-point variables, **a** through **z**. Moreover, it also understands intervals, written

```
(x,y)
```

where **x** is less than or equal to **y**. There are 26 interval valued variables **A** through **Z** that may also be used. Assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as double precision values. This structure is given a type name, **INTERVAL**, by using **typedef**. The **yacc** value stack can also contain floating-point scalars and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of **YYERROR** to handle the following error conditions: division by an interval containing 0 and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc:** 18 shift/reduce and 26 reduce/reduce. The problem can be seen by looking at the two input lines:

```
2.5 + (3.5-4.)
```

and

```
2.5 + (3.5,4.)
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the comma (,) is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is circumvented by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. However, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating-point constants. The C library routine **atof** is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
/* advanced desk calculator parser specification */
%{

#  include  <stdio.h>
#  include  <ctype.h>

typedef struct interval {
      double lo, hi;
      } INTERVAL;

INTERVAL vmul(), vdiv();

double  atof();

double  dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union {
      int ival;
      double dval;
      INTERVAL vval;
      }

%token  <ival>  DREG  VREG    /* indices into dreg, vreg arrays  */

%token  <dval>  CONST         /* floating-point constant  */

%type   <dval>  dexp          /* expression  */

%type   <vval>  vexp          /* interval expression  */
```

```
        /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS    /* precedence for unary minus */

%%

lines : /* empty */
      | lines line
      ;

line  : dexp '\n'
            { printf( "%15.8f\n", $1 ); }
      | vexp '\n'
            { printf( "(%15.8f, %15.8f)\en", $1.lo, $1.hi ); }
      | DREG '=' dexp '\n'
            { dreg[$1] = $3; }
      | VREG '=' vexp '\n'
            { vreg[$1] = $3; }
      | error '\n'
            { yyerrok; }
      ;

dexp : CONST
      | DREG
            { $$ = dreg[$1]; }
      | dexp '+' dexp
            { $$ = $1 + $3; }
      | dexp '-' dexp
            { $$ = $1 -$3; }
      | dexp '*' dexp
            { $$ = $1 * $3; }
      | dexp '/' dexp
            { $$ = $1 /$3; }
      | '-' dexp %prec UMINUS
            { $$ = -$2; }
      | '(' dexp ')'
            { $$ = $2; }
      ;
```

```
vexp   : dexp
            { $$.hi = $$.lo = $1; }
       |'(' dexp ',' dexp ')'
            {
            $$.lo  = $2;
            $$.hi  = $4;
            if( $$.lo  > $$.hi ){
                  printf("interval out of order\n");
                  YYERROR;
                  }
             }
       |VREG
            { $$ = vreg[$1]; }
       |vexp ' + ' vexp
            { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
       |dexp ' + ' vexp
            { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
       |vexp '-' vexp
            { $$.hi = $1.hi -$3.lo;
                  $$.lo = $1.lo -$3.hi; }
       |dexp '-' vexp
            { $$.hi = $1 -$3.lo;
                  $$.lo = $1 -$3.hi;}
       |vexp '*' vexp
            { $$ = vmul( $1.lo, $1.hi, $3 ); }
       |dexp '*' vexp
            { $$ = vmul( $1, $1, $3 ); }
       |vexp '/' vexp
            { if ( dcheck( $3 ) ) YYERROR;
            $$ = vdiv( $1.lo, $1.hi, $3 ); }
       |dexp '/' vexp
            { if ( dcheck( $3 ) ) YYERROR;
            $$ = vdiv( $1, $1, $3 ); }
       |'-' vexp %prec UMINUS
            { $$.hi = -$2.lo; $$.lo = -$2.hi; }
       |'(' vexp ')'
            { $$ = $2; }
       ;

%%

# define BSZ 50 /* buffer size for fp numbers */
```

```
        /* lexical analysis */

yylex(){
        register c;
                        {/* skip over blanks */}
        while( ( c = getchar( ) ) = = ' ' )

        if ( isupper(c) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if ( islower(c) ){
                yylval.ival = c -'a';
                return( DREG );
                }

        if( isdigit( c ) || c = = '.' ){

                /* gobble up digits, points, exponents */

                char buf[BSZ + 1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; + + cp,c = getchar() ){

                        *cp = c;
                        if ( isdigit(c) ) continue;
                        if ( c = = '.' ) {
                          if ( dot + + || exp ) return( '.' );
                              /* above causes syntax error */
                          continue;
                          }

                        if (c = = 'e' ) {
                          if ( exp + + ) return( 'e' );
                              /* above causes syntax error */
                          continue;
                          }

                        /* end of number */
                        break;
                        }
                *cp = '\e0';
                if( (cp-buf) > = BSZ )
                         printf( "constant too long:  truncated\n");
                else ungetc( c, stdin );
                         /* above pushes back last char read */
                yylval.dval = atof ( buf );
                return( CONST );
                }
        return( c );
        }
```

```
INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a; v.lo = b; }
        else { v.hi = b; v.lo = a; }

        if( c>d ) {
                if ( c>v.hi ) v.hi = c;
                if ( d<v.lo ) v.lo = d;
                }
        else {
                if ( d>v.hi ) v.hi = d;
                if ( c<v.lo ) v.lo = c;
                }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return(1);
                }
        return(0);
        }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Old Features

This section mentions synonyms and features supported for historical continuity, but that, for various reasons, are not encouraged.

- Literals may also be delimited by double quotation marks (").

- Literals may be more than one character long. If all the characters are alphabetic, numeric, or underscore, the type number of the literal is defined, just as if the literal did not have the quotation marks around it. Otherwise, it is difficult to find the value for such literals. The use of multicharacter literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job that must be actually done by the lexical analyzer.

- Most places where % is legal, backslash (\) may be used. In particular, the double backslash (\\) is the same as **%%**, **\left** the same as **%left**, etc.

- There are a number of other synonyms:

  %< is the same as **%left**
  %> is the same as **%right**
  %binary and **%2** are the same as **%nonassoc**
  %0 and **%term** are the same as **%token**
  %= is the same as **%prec**

- Actions may also have the form

  ={ ... }

  and the curly braces can be dropped if the action is a single C statement.

- C code between **%{** and **%}** used to be permitted at the head of the rules section, as well as in the declaration section.

intel®

CHAPTER 11
m4: MACRO PROCESSOR

The **m4** macro processor defines and processes specially defined strings of characters called macros. By defining a set of macros to be processed by **m4**, a programming language can be enhanced to make it more structured, more readable, and more appropriate for a particular application.

The **#define** statement in C and the analogous **define** in RATFOR are examples of the basic facility provided by any macro processor--replacement of text by other text.

Besides the straightforward replacement of one string of text by another, **m4** provides

- Macros with arguments
- Conditional macro expansions
- Arithmetic expressions
- File manipulation facilities
- String processing functions

The basic operation of **m4** is copying its input to its output. As the input is read, each alphanumeric token (that is, string of letters and digits) is checked. If the token is the name of a macro, then the name of the macro is replaced by its defining text. The resulting string is reread by **m4**. Macros may also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

m4 provides a collection of about 20 built-in macros. In addition, the user can define new macros. Built-in and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Invoking m4

The invocation syntax for **m4** is

```
m4 [ filename ] . . .
```

Each *filename* argument is processed in order. If there are no arguments, or if an argument is a dash (-), then the standard input is read. The processed text is written to the standard output and can be redirected as in the following example:

```
m4 file1 file2 - >outputfile
```

Note the use of the dash in the above example to indicate processing of the standard input, after the files **file1** and **file2** have been processed by **m4.**

Defining Macros

The primary built-in function of **m4** is **define,** which is used to define new macros. The input

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff.* All subsequent occurrences of *name* will be replaced by *stuff. name* must be alphanumeric and must begin with a letter (the underscore (_) counts as a letter). *stuff* is any text, including text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example

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

defines **N** to be 100 and uses this symbolic constant in a later **if** statement.

The left parenthesis must immediately follow the word **define,** to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis, it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics. For example, in

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

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains three N's.

Names can be defined in terms of other names. For example

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

defines both **M** and **N** to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In **m4**, the latter is true; **M** is 100, so that even if **N** subsequently changes, **M** does not.

This behavior arises because **m4** expands macro names into their defining text as soon as it possibly can. Here, that means that when the string "N" is seen as the arguments of the second **define** statement are being collected, it is immediately replaced by 100; it's just as if you had said

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

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

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

Now **M** is defined to be the string "N", so when you ask for **M** later, you will always get the value of **N** at that time (because the **M** will be replaced by "N" which in turn will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks (` and ') is not expanded immediately but has the quotation marks stripped off. If you say

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

the quotation marks around the "N" are stripped off as the argument is being collected, but they have served their purpose, and "M" is defined as the string "N", not 100. The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word "define" to appear in the output, you have to quote it in the input, as in

```
'define' = 1;
```

As another instance, consider redefining "N":

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

The "N" in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's the same as

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

This statement is ignored by **m4**, since you can only define things that look like names, but it doesn't have the effect you wanted. To truely redefine **N**, you must delay the evaluation by quoting:

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

In **m4**, it is often wise to quote the first argument of a macro.

If the forward and backward quotation marks (` and ') are not convenient for some reason, the quotation marks can be changed with the built-in **changequote**. For example

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

makes the new quotation marks the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-in macros related to **define**. The built-in **undefine** removes the definition of a macro:

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

removes the definition of **N**. Built-in macros can be removed with **undefine**, as in

```
undefine('define')
```

but once you remove one, you can redefine it, but you cannot retrieve the previous definition.

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, pretend that either the word "xenix" or "unix" is defined according to a particular implementation of a program. To perform operations according to which system you have, you might use:

```
ifdef('xenix', 'define(system,1)' )
ifdef('unix', 'define(system,2)' )
```

Don't forget the quotation marks in the above example.

ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('xenix', on XENIX, not on XENIX)
```

Using Arguments

So far we have discussed the simplest form of macro processing--replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$***n* will be replaced by the *n*th argument when the macro is actually used. Thus, the macro **bump**, defined as

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

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have any number of arguments, but only the first nine are accessible, as **$1** to **$9.** (The macro name itself is **$0.**) Arguments not supplied are replaced by null strings, so we can define a macro **cat** that simply concatenates its arguments, like this:

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

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

The arguments **$4** through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

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

defines **a** to be "b c".

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. That is, in

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

there are only two arguments; the second is literally "(b,c)". Note that a bare comma or parenthesis can be inserted by quoting it.

Using Arithmetic Built-In Macros

m4 provides two built-in macros for doing arithmetic on integers. The simplest is **incr,** which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than N, write

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

The more general mechanism for arithmetic is a built-in macro called **eval,** which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

```
unary + and -
** or ^ (exponentiation)
* / % (modulus)
+  -
= =  !=  <  <=  >  >=
!      (not)
& or && (logical and)
| or | | (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1 and of a false relation is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want **M** to have the numeric value 2**N+1. Then we could write:

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

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (e.g., just a number); it usually gives the result you want and is a good habit to get into.

Manipulating Files

You can include a new file in the input at any point by using the built-in macro **include:**

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file.

It is a fatal error if the file named in **include** cannot be accessed. To gain some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it can't access the file.

You can divert the output of **m4** to temporary files during processing and output the collected material upon command. **m4** maintains nine of these diversions, numbered 1 through 9. If you use

divert(n)

all subsequent output is put onto the end of a temporary file referred to as "n". Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. However, you can bring back diversions at any time.

undivert

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text, as does diverting into a diversion with a number not between 0 and 9 inclusive.

The value of **undivert** is not the diverted text. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

Using System Commands

You can run any program in the local operating system with the built-in macro **syscmd.** For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include.**

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp:** a string of the form "XXXXX" in the argument is replaced by the process ID of the current process.

Using Conditionals

The built-in macro **ifelse** performs arbitrary conditional testing. In the simplest form,

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

compares the two strings **a** and **b.** If these are identical, **ifelse** returns the string **c;** otherwise it returns **d.** Thus, we might define a macro called **compare** that compares two strings and returns "yes" or "no" if they are the same or different.

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

Note the quotation marks, which prevent premature evaluation of **ifelse.**

If the fourth argument is missing, it is treated as empty.

ifelse can have any number of arguments and provides a multi-way decision capability. In the input

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

if the string **a** matches the string **b,** the result is **c.** Otherwise, if **d** is the same as **e,** the result is **f.** Otherwise the result is **g.** If the final argument is omitted, the result is null, so

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

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

Manipulating Strings

The built-in **len** returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and

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

is 5.

The built-in **substr** can be used to produce substrings of strings. For example

```
substr(s, i, n)
```

returns the substring of **s** that starts at position **i** (origin zero) and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

```
substr('now is the time', 1)
```

is

```
ow is the time
```

The command

```
index(s1,s2)
```

returns the index (position) in **s1** where the string **s2** occurs, or -1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

```
translit(s, f, t)
```

modifies **s** by replacing any character of **s** found in **f** by the corresponding character of **t**. That is

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

replaces the vowels in **s** with the corresponding digits. If **t** is shorter than **f**, characters that don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

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

deletes vowels from **s**.

The built-in macro **dnl** deletes all characters that follow it up to and including the next newline. It is useful mainly for deleting empty lines that otherwise tend to clutter up **m4** output. For example, if you specify

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

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this is

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

Printing

The built-in macro **errprint** writes its arguments out on the standard error file. Thus, you can write

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

The built-in macro **dumpdef** is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, all definitions are output. Otherwise, only the definitions of the terms given as arguments to **dumpdef** are output. The arguments to **dumpdef** should be quoted.

intel®

APPENDIX A
C LANGUAGE PORTABILITY

The standard definition of the C programming language leaves many details to be decided by individual implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP-11), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment, which is determined by the system calls and library routines it uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small 8-bit microprocessors to large nainframes. This appendix is concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all XENIX systems to date run on hardware with the following basic characteristics:

- ASCII character set
- 8-bit bytes
- 2-byte or 4-byte integers
- Two's complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output are performed. This is left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. These are described briefly in a later section.

This appendix is not intended as a C language primer. It is assumed that the reader is familiar with C and with the basic architecture of common microprocessors.

Program Portability

A program is portable if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. The first is to avoid using inherently nonportable language features. The second is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coding path names unless a path name is common to all systems (e.g., **/etc/passwd**).

Files required at compile time (i.e., include files) may also introduce nonportability if the path names are not the same on all machines. In some cases, include files containing machine parameters can be used to make the source code itself portable.

Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered on XENIX systems.

Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is exactly an 8-bit byte.

Word Length

In C, the size of the basic data types for a given implementation are not formally defined. Thus they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that no assumptions are portable. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's complement machine) can be obtained with

```
#define MAXPOS ((int)(((unsigned)-1) >> 1))
```

This is preferable to something like:

```
#ifdef i8086
#define MAXPOS 32767
#else
      ...
#endif
```

To find the number of bytes in an **int**, use **sizeof(int)** rather than 2, 4, or some other nonportable constant.

Storage Alignment

The C language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPUs, such as the PDP-11 and M68000, require that data types longer than one byte be aligned on even-byte address boundaries. Others, such as the 8086, 80286, and VAX-11 have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses, or even long word addresses. Thus, on the VAX-11, the following code sequence gives 8, even though the VAX hardware can access an **int** (a 4-byte word) on any physical starting address:

```
struct s_tag {
      char c;
int i;
}; printf("%d\n",sizeof(struct s_tag));
```

The principal implications of this variation in data storage are that data accessed as nonprimitive data types is not portable, and code that makes use of knowledge of the layout on a particular machine is not portable.

Thus unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used simply to have different data in the same space at different times. For example, if the following union were used to obtain 4 bytes from a long word, the code would not be portable:

```
union {
      char c[4];
      long lw;
} u;
```

The **sizeof** operator should always be used when reading and writing structures:

```
struct s_tag st;

...

write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does not produce a portable data file. Portability of data is discussed in a later section.

Note that the **sizeof** operator returns the number of bytes an object would occupy in an array. Thus on machines where structures are always aligned to begin on a word boundary in memory, the **sizeof** operator will include any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs whether or not the argument is actually an array element.

Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example, some XENIX/UNIX systems have an include file **misc.h** that contains the following structure declaration:

```
/*
* structure to access an
* integer in bytes
*/
struct {
        char        lobyte;
        char        hibyte;
};
```

With certain less restrictive compilers this could be used to access the high and low order bytes of an integer separately, and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

Note that even this operation is only applicable to machines with two bytes in an **int**.

One result of the byte ordering problem is that the following code sequence will not always perform as intended:

```
int  c = 0;

read(fd, &c, 1);
```

On machines where the low order byte is stored first, the value of **c** will be the byte value read. On other machines the byte is read into some byte other than the low order one, and the value of **c** is different.

Bitfields

Bitfields are not implemented in all C compilers. When they are, no field may be larger than an **int**, and no field can overlap an **int** boundary. If necessary the compiler will leave gaps and move to the next **int** boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an **int**. Thus, while bitfields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

To ensure portability, no individual field should exceed 16 bits.

Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to nonportable pointer operations. The **lint** program is particularly useful for detecting questionable pointer assignments and comparisons.

The common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the given array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The **lint** program will issue warning messages about such uses of pointers. Code like this is very rarely necessary or valid. It is acceptable, however, when using the **malloc** function to allocate space for variables that do not have **char** type. The routine is declared as type **char *** and the return value is cast to the type to be stored in the allocated memory. If this type is not **char *** then **lint** will issue a warning concerning illegal type conversion. In addition, the **malloc** function is written to always return a starting address suitable for storing all types of data. **lint** does not know this, so it gives a warning about possible data alignment problems, too. In the following example, **malloc** is used to obtain memory for an array of 50 integers.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50);
```

This example will cause a warning message from **lint.**

The C reference manual (contained in *The C Programming Language* by Kernighan and Ritchie) states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the **int** type depends on the given machine and implementation. This type is **long** on some machines and **short** on others. In general, do not assume that **sizeof(char *) == sizeof(int).**

In most implementations, the null pointer value **NULL** is defined to be the integer value 0. This can lead to problems for functions that expect pointer arguments larger than integers. For portable code, always use

```
func( (char *)NULL );
```

to pass a **NULL** value of the correct size.

Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11, only 64K bytes may be available for program and data combined. Larger PDP-11's and some 16-bit microprocessors allow 64K bytes of data and 64K bytes of program text. Other machines, such as the 80286, allow considerably more text, and possibly more data as well.

Large programs or programs that require large data areas may have portability problems on small machines.

Character Set

The C language does not require the use of the ASCII character set. In fact, the only character set requirements are that all characters must fit in the **char** data type and all characters must have positive values.

In the ASCII character set, all characters have values from 0 to 127. Thus they can all be represented in 7 bits, and on an 8-bits-per-byte machine are all positive, whether **char** is treated as signed or unsigned.

XENIX defines a set of macros in the header file **/usr/include/ctype.h** that should be used for most tests on character quantities. They provide insulation from the internal structure of the character set and in most cases their names are more meaningful than the equivalent line of code. Compare

```
if(isupper(c) )
```

to

```
if( (c >= 'A') && (c <= 'Z') )
```

With some of the other macros, such as **isdigit** to test for a hex digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an **if** statement.

Compiler Differences

A number of C compilers are available for various XENIX systems. The "Ritchie" compiler is available for PDP-11 systems. Available for the PDP-11 and most other XENIX systems is the Portable C Compiler.

Signed/Unsigned char, Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory.

The sign extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

Shift Operations

The left shift operator << shifts its operand a number of bits left, filling vacated bits with zero. This is a logical shift. The right shift operator >>, when applied to an unsigned quantity, performs a logical shift operation. When applied to a signed quantity, the vacated bits may be filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that uses knowledge of a particular implementation is nonportable.

The PDP-11 compilers use arithmetic right shift. To avoid sign extension, you must shift and mask out the appropriate number of high order bits:

```
char c;

c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using the divide operator:

```
char c;

c = c / 8;
```

Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

The loader used may also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

On some non-XENIX C implementations, uppercase and lowercase letters are not distinct in identifiers.

Register Variables

The number and type of register variables in a function depends on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. Typically, up to three register declarations are significant, and they must be of type **int, char,** or pointer. While other machines and compilers may support declarations such as

```
register unsigned short
```

this should not be relied on.

Since the compiler ignores excess variables of register type, the most important register type variables should be declared first. Thus, if any are ignored, they will be the least important ones.

Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int.**

For example

```
char c;

if(c = = 0x80)
        ...
```

will never evaluate true on a machine that sign extends since **c** is sign extended before the comparison with 0x80, an **int.**

The only safe comparison between **char** type and an **int** is the following:

```
char c;

if(c = = 'x')
        ...
```

This is reliable because C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example, the following program prints "ff80" on a PDP-11:

```
main()
{
        printf("%x\n", '\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int.** Machines that sign extend **char** can produce unexpected results. For example, the following program gives -128 on some machines:

```
char c = 128;
printf("%d\n",c);
```

This is because **c** is converted to **int** before being passed to the function. The function itself has no knowledge of the original type of the argument and is expecting an **int.** The correct way to handle this is to code defensively and allow for the possibility of sign extension:

```
char c = 128;
printf("%d\n", c & 0xff);
```

Functions with Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases, the code depends on the size of various data types.

XENIX has an include file, **/usr/include/varargs.h,** that contains macros for use in variable argument functions to access the arguments in a portable way:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) ((mode *)(list += sizeof(mode)))[-1]
```

The **va_end()** macro is not currently required. Use of the other macros will be demonstrated by an example of the **fprintf** library routine. This has a first argument of type **FILE *** and a second argument of type **char *.** Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument 2.

The first few lines of **fprintf** to declare the arguments and find the output file and control string address could be

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

int
fprintf(va_alist)
va_dcl;
{
        va_list ap;        /* pointer to arg list     */
        char *format;
        FILE *fp;

        va_start(ap);     /* initialize arg pointer */
        fp = va_arg(ap, (FILE *));
        format = va_arg(ap, (char *));

                ...
}
```

Note that just one argument is declared to **fprintf.** This argument is declared by the **va_dcl** macro to be type **int,** although its actual type is unknown at compile time. The argument pointer **ap** is initialized by **va_start** to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the **va_arg** macro. This has a type as its second argument, and this controls what data is removed from the stack, and how far the argument pointer **ap** is incremented. In **fprintf,** once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to **va_arg().** For example, arguments of type **double, int *,** and **short** could be retrieved as follows:

```
double dint;
int *ip;
short s;

dint = va_arg(ap, double);
ip = va_arg(ap, (int *));
s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular, no holes must be left by the compiler, and types smaller than **int** (e.g., **char** and **short** on long word machines) must be declared as **int.**

Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Thus

```
func(i + +, i + +);
```

is extremely nonportable, and even

```
func(i + +);
```

is unwise if **func** is ever likely to be replaced by a macro, since the macro may use **i** more than once. Certain XENIX macros are commonly used in user programs; these are all guaranteed to use their argument once, and so can safely be called with a side-effect argument. The most common examples are **getc, putc, getchar,** and **putchar.**

Operands to the following operators are guaranteed to be evaluated left to right:

```
,    &&    ||    ?    :
```

Note that the comma operator above is a separator for C expressions. (For example, the expression (a, b, c, d) evaluates a, then b, then c, and last d, returning the value of d as the value of the entire expression.) A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Thus the declaration

```
register int a, b, c, d;
```

on a PDP-11 where only three register variables may be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
register int a;
register int b;
register int c;
register int d;
```

Program Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable, and those that particularly aid portability.

We are concerned here primarily with portability under the XENIX operating system. Many of the XENIX system calls are specific to that particular operating system environment and are not present on all other operating system implementations of C. Examples of this are **getpwent** for accessing entries in the XENIX password file, and **getenv,** which is specific to the XENIX concept of a process's environment.

Any program containing hard-coded path names to files or directories, or user IDs, login names, terminal lines, or other system dependent parameters is nonportable. These types of constants should be in header files, passed as command line arguments, obtained from the environment, or obtained by using the XENIX default parameter library routines **defopen** and **defread.**

Within XENIX, most system calls and library routines are portable across different implementations and XENIX releases. However, a few routines have changed in their user interface. The XENIX library routines are usually portable among XENIX systems.

Note that the members of the **printf** and **scanf** families, **printf, fprintf, scanf, fscanf,** and **sscanf** have changed in several ways during the evolution of XENIX, and some features are not completely portable. The return values of these routines cannot be relied on to have the same meaning on all systems. Some of the format conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers, and the specification of **long** integers on 16-bit word machines.

Portability of Data

Data files are almost always nonportable across different machine CPU architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data file portability is to write and read data files as one-dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters and interpreted that way. Thus ASCII text files can usually be moved between different machine types without too many problems.

lint

lint is a C program checker that attempts to detect features of a collection of C source files that are nonportable or even incorrect C. One particular advantage of **lint** over any compiler checking is that **lint** checks function declaration and usage across source files. Neither compiler nor loader do this.

lint will generate warning messages about nonportable pointer arithmetic, assignments, and type conversions. However, being passed by **lint** is not a guarantee that a program is completely portable.

Byte Ordering Summary

The following conventions are used in Tables A-1 and A-2 below:

a0 The lowest physically addressed byte of the data item. a1 has a byte address a0 + 1, and so on.

b0 The least significant byte of the data item, b1 being the next least significant, and so on.

Note that any program that actually makes use of the following information is guaranteed to be nonportable.

Table A-1. Byte Ordering for Short Types

CPU	Byte Order	
	a0	a1
8086	b0	b1
80286	b0	b1
PDP-11	b0	b1
VAX-11	b0	b1
M68000	b1	b0
Z8000	b1	b0

Table A-2. Byte Ordering for Long Types

CPU	Byte Order			
	a0	a1	a2	a3
8086	b0	b1	b2	b3
80286	b0	b1	b2	b3
PDP-11	b2	b3	b0	b1
VAX-11	b0	b1	b2	b3
M68000	b3	b2	b1	b0
Z8000	b3	b2	b1	b0

intel®

APPENDIX B
PROGRAMMING COMMANDS

This section describes the programming commands available in the XENIX 286 Extended System product.

Syntax

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*option*] ... [*cmdarg*] ...

where:

name The file name or path name of an executable file.

option A single letter representing a command option. By convention, most options are preceded by a hyphen. Option letters can sometimes be grouped together, as in **-abcd.** Alternatively they are specified individually, as in **-a -b -c -d.** The method of specifying options depends on the syntax of the command. Some options require arguments. For example, the **-f** option for many commands often takes a following file name argument.

cmdarg A path name or other command argument not beginning with a hyphen. It may also be a hyphen alone by itself, indicating the standard input.

See Also

getopt in "Commands" in the *XENIX 286 Reference Manual*

getopt in "System Functions" in the *XENIX 286 C Library Guide*

Diagnostics

Upon termination, each command returns 2 bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program. (See **wait** and **exit** in "System Functions" in the *XENIX 286 C Library Guide.*) The byte supplied by the system is zero for normal termination; the byte supplied by the program is customarily zero to indicate successful execution and nonzero to indicate troubles such as erroneous parameters or bad or inaccessible data. It is called variously "exit code," "exit status," or "return code," and is described only where special conventions are involved.

Notes

Not all commands adhere to the above syntax.

adb - Invokes a general-purpose debugger.

Syntax

adb [-w] [-p prompt] [objfil [corefile]]

Description

A general-purpose debugging program, **adb** may be used to examine files and to provide a controlled environment for the execution of XENIX programs.

Normally an executable program file, *objfil* preferably contains a symbol table; if not, then the symbolic features of **adb** cannot be used, although the file can still be examined. The default for *objfil* is **a.out**. *corefile* is assumed to be a core image file produced after executing *objfil*. The default for *corefile* is **core**.

Requests to **adb** are read from the standard input, and responses are written to the standard output. If the **-w** option is present, then both *objfil* and *corefile* are created if necessary and opened for reading and writing so that files can be modified with **adb**. The QUIT and INTERRUPT keys cause **adb** to return to the next command. The **-p** option defines the prompt string. It may be any combination of characters. The default is an asterisk (*).

In general requests to **adb** are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present, then dot is set to *address*. Initially, dot is set to zero. For most commands, *count* specifies how many times the command will be executed. The default count is 1. A special expression, *address* has the form

[*segment*] *offset*

where *segment* gives the address of a specific text or data segment, and *offset* gives an offset from the beginning of that segment. If *segment* is not specified, then the last segment value in a command is used.

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged, then addresses are interpreted in the usual way in the address space of the subprocess. For details on address mapping, see the "Addresses" section later in this entry.

Expressions

. The value of dot.

+ The value of dot incremented by the current increment.

^ The value of dot decremented by the current increment.

" The last address typed.

integer
An octal number if *integer* begins with a **0**; a hexadecimal number if preceded by **#** or **0x**; otherwise a decimal number.

integer.fraction
A 32-bit floating-point number.

'cccc'
The ASCII value of up to 4 characters. The backslash (\) may be used to escape an apostrophe (').

<name
The value of *name*, which is either a variable name or a register name. **adb** maintains a number of variables (see "Variables" later in this entry) named by single letters or digits. If *name* is a register name, then the value of the register is obtained from the system header in *corefile*. The register names are **ax, bx, cx, dx, di, si, bp, fl, ip, cs, ds, ss, es, sp.** The name **fl** refers to the status flags.

symbol
A *symbol* is a sequence of uppercase or lowercase letters, underscores, or digits not starting with a digit. The value of *symbol* is taken from the symbol table in *objfil*. An initial underscore (_) or tilde (~) will be prepended to *symbol* if needed.

_symbol
In C, the "true name" of an external symbol begins with _. It may be necessary to use this name to distinguish it from internal or hidden variables of a program.

(*exp*)
The value of the expression *exp*.

Monadic Operators

******exp* The contents of the location addressed by *exp*.

-exp Integer negation.

~exp Bitwise complement.

Dyadic Operators

Dyadic operators are left-associative and are less binding than monadic operators.

e1 + *e2* Integer addition

e1 - *e2* Integer subtraction

e1 * *e2* Integer multiplication

e1 % *e2* Integer division

e1 & *e2* Bitwise AND

e1 | *e2* Bitwise OR

e1 ^ *e2* Remainder after division of *e1* by *e2*.

e1 # *e2* Value of *e1* rounded up to the next multiple of *e2*.

Commands

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands **?** and **/** may be followed by *****; see the "Addresses" section later in this entry for details.)

?*f* Locations starting at *address* in *objfil* are printed according to the format *f*.

/*f* Locations starting at *address* in *corefile* are printed according to the format *f*.

=f The value of *address* itself is printed in the styles indicated by the format *f*. (For **i** format, **?** is printed for the parts of the instruction that reference subsequent words.)

A format consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format, dot is incremented temporarily by the amount given for each format letter. If no format is given, then the last format is used. The format letters available are as follows:

o 2 Prints 2 bytes in octal. All octal numbers output by **adb** are preceded by 0.

O 4 Prints 4 bytes in octal.

q 2 Prints in signed octal.

Q 4 Prints long signed octal.

d 2 Prints in decimal.

D 4 Prints long decimal.

x 2 Prints 2 bytes in hexadecimal.

X 4 Prints 4 bytes in hexadecimal.

u 2 Prints as an unsigned decimal number.

U 4 Prints long unsigned decimal.

f 4 Prints the 32-bit value as a floating-point number.

F 8 Prints double floating point.

b 1 Prints the addressed byte in octal.

c 1 Prints the addressed character.

C 1 Prints the addressed character using the following escape convention. Character values 000 to 040 are printed as an at sign (@) followed by the corresponding character in the octal range 0100 to 0140. The at sign character itself is printed as **@@**.

s n Prints the addressed characters until a zero character is reached.

S n Prints a string using the at sign (@) escape convention. Here, **n** is the length of the string including its zero terminator.

Y 4 Prints 4 bytes in date format. (See **ctime** in "Svstem Functions" in the *XENIX 286 C Library Guide.*)

i n Prints as machine instructions. **n** is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.

a 0 Prints the value of dot in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

A 0 Prints the value of dot in absolute form.

p 2 Prints the addressed value in symbolic form using the same rules for symbol lookup as **a**.

t 0 Tabs to the next appropriate tab stop when preceded by an integer. For example, **8t** moves to the next 8-space tab stop.

r 0 Prints a space.

n 0 Prints a newline.

'*string***' 0**
Prints the enclosed *string*.

^ Decrements dot by the current increment. Nothing is printed.

+ Increments dot by 1. Nothing is printed.

- Decrements dot by 1. Nothing is printed.

newline
If the previous command temporarily incremented dot, makes the increment permanent. Repeat the previous command with a count of 1.

[?/]l *value* [*mask*]
Words starting at dot are masked with *mask* and compared with *value* until a match is found. If **L** is used, then the match is for 4 bytes at a time instead of 2. If no match is found, then dot is unchanged; otherwise, dot is set to the matched location. If *mask* is omitted, then -1 is used.

[?/]w *value* ...
Writes the 2-byte value into the addressed location. If the command is **W**, writes 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]m *segnum fpos* [*size*]
Sets new values for the given segment's file position and size. If *size* is not given, then only the file position is changed. *segnum* must be the segment number of a segment already in the memory map. (See the "Addresses" section later in this entry.) If **?** is given, a text segment is affected; if **/** is given, a data segment is affected.

[?/]**M** *segnum fpos size*
Creates a new segment in the memory map. The segment is given a file position *fpos* and physical size *size*. *segnum* must not already exist in the memory map. If **?** is given, a text segment is created; if **/** is given, a data segment is created.

>name
dot is assigned to the variable or register named.

! A shell is called to read the rest of the line following **!**.

$*modifier*
Miscellaneous commands. The available modifiers are:

<f Read commands from the file *f* and return.

>f Send output to the file *f*, which is created if it does not exist.

r Print the general registers and the instruction addressed by **ip.** dot is set to **ip.**

f Print the floating registers in single or double length.

b Print all breakpoints and their associated counts and commands.

c C stack backtrace. If *address* is given, then it is taken as the address of the current frame (instead of **bp**). If C is used, then the names and (16-bit) values of all automatic and static variables are printed for each active function. If *count* is given, then only the first *count* frames are printed.

e The names and values of external variables are printed.

w Set the page width for output to *address* (default 80).

s Set the limit for symbol matches to *address* (default 255).

o Sets input and output default format to octal.

d Sets input and output default format to decimal.

x Sets input and output default format to hexadecimal.

q Exit from **adb.**

v Print all nonzero variables in octal.

m Print the address map.

:*modifier*

Manage a subprocess. Available modifiers are

brc Set breakpoint at *address*. The breakpoint is executed *count*-1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets dot to zero, then the breakpoint causes a stop.

dl Delete breakpoint at *address*.

r[*arguments*]
Run *objfil* as a subprocess. If *address* is given explicitly, then the program is entered at this point; otherwise, the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on upon entry to the subprocess.

R[*arguments*]
Same as the **r** command except that *arguments* are passed through a shell before being passed to the program. This means shell metacharacters can be used in file names.

cos The subprocess is continued and signal s is passed to it. (See the entry for **signal** in "System Functions" in the *XENIX 286 C Library Guide*.) If *address* is given, then the subprocess is continued at this address. If no signal is specified, then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

ss Same as **c** except that the subprocess is single-stepped *count* times. If there is no current subprocess then *objfil* is run as a subprocess as for **r**. In this case, no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

Variables

adb provides a number of variables. Named variables are set initially by **adb** but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry, the following are set from the system header in the core file. If *corefile* does not appear to be a core file then these values are set from *objfil*:

d The data segment size.
e The entry point.
m The execution type.
n The number of segments.
t The text segment size.

Addresses

Addresses in **adb** refer to either a location in a file or in actual memory. When there is no current process in memory, **adb** addresses are computed as file locations, and requested text and data are read from the *objfil* and *corefile* files. When there is a process, such as after a **:r** command, addresses are computed as actual memory locations.

All text and data segments in a program have associated memory map entries. Each entry has a unique segment number. In addition, each entry has the file position of that segment's first byte and the physical size of the segment in the file. When a process is running, a segment's entry has a virtual size that defines the size of the segment in memory at the current time. This size can change during execution.

When an address is given and no process is running, the file location corresponding to the address is calculated as

effective-file-address = file-position + offset

If a process is running, the memory location is simply the offset in the given segment. These addresses are valid if and only if

0 <= offset <= size

where **size** is physical size for file locations and virtual size for memory locations. Otherwise, the requested address is not legal.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, file position is set to 0, and size is set to the maximum file size. In this way, the whole file can be examined with no address translation.

So that **adb** may be used on large files, all appropriate values are kept as signed 32-bit integers.

Files

/dev/mem
/dev/swap
a.out
core

See Also

ptrace in "System Functions" in the *XENIX 286 C Library Guide*

a.out, core in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

The message "adb" appears when there is no current command or format.

Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. may appear.

Exit status is zero unless last command failed or returned nonzero status.

Notes

A breakpoint set at the entry point is not effective on initial entry to the program.

System calls cannot be single-stepped.

Local variables whose names are the same as an external variable may foul up the accessing of the external variable.

admin - Creates and administers SCCS files.

Syntax

admin [-n] [-i[*name*]] [-r*rel*] [-t[*name*]] [-f*flag*[*flag-val*]]
[-d*flag*[*flag-val*]] [-a*login*] [-e*login*] [-m[*mrlist*]]
[-y[*comment*]] [-h] [-z] *file*...

Description

admin is used to create new SCCS (source code control system) files and change parameters of existing ones. Arguments to **admin** may appear in any order. They consist of options, which begin with a dash (-), and named files. (Note that SCCS file names must begin with the characters "s.".) If a named file does not exist, it is created, and its parameters are initialized according to the specified options. Parameters not initialized by an option are assigned a default value. If a named file does exist, parameters corresponding to specified options are changed, and other parameters are left as is.

If a directory is named, **admin** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (the last component of whose path names does not begin with **s.**) and unreadable files are silently ignored. If the dash is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The options are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

-n This option indicates that a new SCCS file is to be created.

-i[*name*]
The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file. (See the description of the **-r** option, which follows, for the delta numbering scheme.) If the **i** option is used but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this option is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an **admin** command on which the **-i** option is supplied. Using a single **admin** to create two or more SCCS files requires that they be created empty (no **-i** option). Note that the **-i** option implies the **-n** option.

-r*rel*
The release into which the initial delta is inserted. This option may be used only if the **-i** option is also used. If the **-r** option is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1. (By default, initial deltas are named 1.1.)

-t[*name*]

The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **-t** option is used and **admin** is creating a new SCCS file (the **-n** and/or **-i** options are also used), the descriptive text file name must also be supplied. In the case of existing SCCS files, a **-t** option without a file name causes removal of descriptive text (if any) currently in the SCCS file, and a **-t** option with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

-f*flag*[*flag-val*]

This option specifies a *flag*, and possibly a value for the *flag*, to be placed in the SCCS file. Several **f** options may be supplied on a single **admin** command line. The allowable *flag* and *flag-val* combinations are

b Allows use of the **-b** option on a **get** command to create branch deltas.

c*ceil* The highest release (or "ceiling"), a number less than or equal to 9999, that may be retrieved by a **get** command for editing. The default value for an unspecified **c** flag is 9999.

f*floor*

The lowest release (or "floor"), a number greater than 0 but less than 9999, that may be retrieved by a **get** command for editing. The default value for an unspecified **f** flag is 1.

d*SID* The default delta number (SID, or SCCS identification) to be used by a **get** command.

i Causes the "No id keywords (ge6)" message issued by **get** or **delta** to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see **get** later in this appendix) are found in the text retrieved or stored in the SCCS file.

j Allows concurrent **get** commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to which deltas can no longer be made. (A **get -e** against one of these "locked" releases fails.) A *list* is one or more list items separated by commas. A list item is either a release number or the letter "a" (to indicate all releases for the file).

n Causes **delta** to create a "null" delta in each release being skipped (if any) when a delta is made in a new release. For example, in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped. These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file, thus preventing branch deltas from being created from them in the future.

q*text*
User-definable text substituted for all occurrences of the keyword in SCCS file text retrieved by **get.**

m*mod*
Module name of the SCCS file substituted for all occurrences of the **admin** keyword in SCCS file text retrieved by **get.** If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type*
type of module in the SCCS file substituted for all occurrences of keyword in SCCS file text retrieved by **get.**

v[pgm]
Causes **delta** to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program. (See the entry for **delta** later in this appendix.) If this flag is set when creating an SCCS file, the **m** option must also be used even if its value is null.

-d[*flag*]
Causes removal (deletion) of the specified *flag* from an SCCS file. The **-d** option may be specified only when processing existing SCCS files. Several **-d** options may be supplied on a single **admin** command. See the **-f** option for allowable flag *names.*

l*list* A *list* of releases to be "unlocked". See the **-f** option for a description of the **l** flag and the syntax of a list.

-a*login*
A login name, or numerical XENIX group ID, to be added to the list of users who may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all login names common to that group ID. Several **-a** options may be used on a single **admin** command line. As many logins, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

-e*login*
A login name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all login names common to that group ID. Several **-e** options may be used on a single **admin** command line.

-**y**[*comment*]
The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of **delta.** Omission of the -**y** option results in a default comment line being inserted in the form

```
YY/MM/DD HH:MM:SS by login
```

The -**y** option is valid only if the -i and/or -**n** options are specified (that is, a new SCCS file is being created).

-**m**[*mrlist*]
The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to **delta.** The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or if the MR validation fails.

-h Causes **admin** to check the structure of the SCCS file (see **sccsfile** in "File Formats" in the *XENIX 286 C Library Guide*) and to compare a newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This option inhibits writing on the file, thereby nullifying the effect of any other options supplied, and is therefore only meaningful when processing existing files.

-**z** The SCCS file checksum is recomputed and stored in the first line of the SCCS file. (See -h, the preceding description.) Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

Files

The last component of all SCCS file names must be of the form **s.***filename*. New SCCS files are created read-only (0444 modified by **umask**--see **chmod** in "Commands" in the *XENIX 286 Reference Manual*). Write permission in the pertinent directory is, of course, required to create a file. All writing done by **admin** is to a temporary x-file, called **x.***filename* (see **get** later in this appendix), which is created with read-only if the **admin** command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of **admin**, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 0755 and that SCCS files themselves be read-only. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 0644 by the owner allowing use of a text editor. Care must be taken. The edited file should always be processed by an **admin** -h to check for corruption followed by an **admin** -**z** to generate a proper checksum. Another **admin** -h is recommended to ensure that the SCCS file is valid.

admin also makes use of a transient lock file called **z.***filename*, which is used to prevent simultaneous updates to the SCCS file by different users. See **get** for further information.

See Also

delta, get, help, prs

ed, what in "Commands" in the *XENIX 286 Reference Manual*

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Refer to **help** for explanations.

ar - Maintains archives and libraries.

Syntax

ar *key* [*posname*] *afile name*...

Description

ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor, though it can be used for any similar purpose.

key is one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcl**. *afile* is the archive file. Each *name* is a constituent file in the archive file. The *key* characters have these meanings:

d Deletes the named files from the archive file.

r Replaces the named files in the archive file. If the optional character **u** is used with **r**, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then the *posname* argument must be present; it specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise, new files are placed at the end.

q Quickly appends the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece by piece.

t Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Prints the named files in the archive.

m Moves the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r**, it specifies where the files are to be moved.

x Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v Verbose. Under the verbose option, **ar** gives a file-by-file description of the creation of a new archive file from the old archive and the constituent files. When used with **t**, it gives a long listing of all information about the files. When used with **x** or **p**, it precedes each file with a name.

c Create. Normally **ar** will create *afile* when it needs to. The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally **ar** places its temporary files in the directory **/tmp**. This option causes them to be placed in the local directory.

Files

/tmp/v* Temporary files

See Also

ld, lorder

ar in "File Formats" in the *XENIX 286 C Library Guide*

Notes

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

as - Invokes the XENIX assembler.

Syntax

as [*options*] *file*.s

Description

as is the XENIX assembler. It reads and assembles 8086/286 assembly language instructions from the source file named *file*.**s** and creates either a linkable object file named *file*.**o** or an executable program named **a.out**. The extension **.s** is recommended but not required. If this extension is not given, **as** displays a warning and continues processing.

These options are available:

-l Creates an assembly listing file called *file*.**L**. This file lists the source instructions, the assembled (binary) code for each instruction, and any assembly errors.

-nl *num*
Sets the maximum length of external symbols to *num*. Names longer than *num* are truncated before being copied to the external symbol table.

-g Directs the assembler to interpret undefined symbols as globally-defined external symbols. If not given, undefined symbols cause an assembly error.

-Mm Creates a middle model object file suitable for linking with other middle model object files. The resulting text segment is named *FILE*_**TEXT**, where *FILE* is the *file* argument, but in uppercase letters.

-NT *name*
Sets the text segment name of the assembled code to *name*. This option overrides the default text segment.

-NM *name*
Sets the module name of the assembled code to *name*. The option overrides the default module name.

-o *objfile*
Copies the assembled instructions to the file named *objfile*. This file is executable only if no errors occur during the assembly. This option overrides the default object name.

Files

/bin/as

See Also

cc, ld

a.out in "File Formats" in the *XENIX 286 C Library Guide*

cb - Beautifies C programs.

Syntax

cb [*file*]

Description

cb takes a copy of the C program in *file* and places it on the standard output with spacing and indentation that displays the structure of the program. If *file* is not given, it acts on the standard input.

cc - Invokes the C compiler.

Syntax

cc [*options*] *filename*...

Description

cc is the XENIX C compiler command. It creates executable programs by compiling and linking the files named by the *filename* arguments. **cc** copies the resulting program to the file **a.out.**

The *filename* extension can name any C or assembly language source file or any object or library file. C source files must have a **.c** file name extension. Assembly language source files must have **.s,** object files **.o,** and library files **.a** extensions.

cc invokes the C compiler for each C source file and copies the results to an object file with a base name that is the same as the source file but with an extension of **.o. cc** invokes the XENIX assembler, **as,** for each assembly source file and copies the results to an object file with extension **.o. cc** ignores object and library files until all source files have been compiled or assembled. It then invokes the XENIX link editor, **ld,** and combines all the object files it has created together with object files and libraries given in the command line to form a single program.

Files are processed in the order in which they are encountered in the command line, so the order of files is important. Library files are examined only if functions referenced in previous files have not yet been defined. Library files must be in **ranlib** format; that is, the first member must be named **_.SYMDEF,** which is a dictionary for the library. The library is searched repeatedly to satisfy as many references as possible. Only those functions that define unresolved references are concatenated. A number of "standard" libraries are searched automatically. These libraries support the standard C functions and program startup routines. The libraries used depend on the program's memory model. (See the "Memory Models" section later in this entry.) The entry point of the resulting program is set to the beginning of the program function **main.**

These options are available:

-P Preprocesses each source file and copies the results to a file with a base name that is the same as the source file's but with an extension of **.i.** Preprocessing performs the actions specified by the preprocessing directives.

-E Preprocesses each source file as described for **-P,** but copies the result to the standard output. The option also places a **#line** directive with the current input line number and source file name at the beginning of output for each file.

-C Preserves comments when processing a file with **-E** or **-P**. That is, comments are not removed from the preprocessed source. This option may be used only in conjunction with **-E** or **-P**.

-D *name*[=*string*]
Defines *name* to the preprocessor as though defined by **#define** in each source file. The form **-D***name* sets *name* to 1. The form **-D***name*=*string* sets *name* to the given *string*.

-I *pathname*
Adds *pathname* to the list of directories to be searched when an **#include** file is not found in the directory containing the current source file or whenever angle brackets (< >) enclose the file name. If the file name cannot be found in directories in this list, directories in a standard list are searched.

-X Removes the standard directories from the list of directories to be searched for **#include** files.

-V *string*
Copies *string* to the object file created from the given source file. This option is often used for version control.

-W *num*
Sets the output level for compiler warning messages. If *num* is 0, no warning messages are issued. If *num* is 1, only warnings about program structure and overt typing mismatches are issued. If *num* is 2, warnings about strong typing mismatches are issued. If *num* is 3, warnings for all automatic conversions are issued. This option does not affect complier error message output.

-w Prevents compiler warning messages from being issued. Same as **-W 0**.

-p Adds code for program profiling. Profiling code counts the number of calls to each routine in the program and copies this information to the **mon.out** file. This file can be examined by using the **prof** command. (See the **prof** entry later in this appendix.)

-i Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are all allocated in separate physical segments. The text portion will be read-only and may be shared by all users executing the file. The option is implied when creating middle or large model programs.

-F *num*
Sets the size of the program stack to *num* bytes. Default stack size, if not given, is 4 Kbytes.

-K Removes stack probes from a program. Stack probes are used to detect stack overflow on entry to program routines.

-nl *num*
Sets the maximum length of external symbols to *num*. Names longer than *num* are truncated before being copied to the external symbol table.

-M *string*

Sets the program configuration. This configuration defines the program's memory model, word order, and data threshold. It also enables C langauge enhancements such as advanced instruction set and keywords. The string may be any combination of the following; however, **s, m,** and **l** are mutually exclusive:

s Creates a small model program (default).
m Creates a middle model program.
l Creates a large model program.
e Enables the far and near keywords.
2 Enables 286 code generation for compiled C source files.
b Reverses the word order for long types so that the high-order word is first. (The default puts the low-order word first.)
t *num*
Sets the size of the largest data item in the group to *num*. Default is 32,767.

-c Creates a linkable object file for each source file but does not link these files. No executable program is created.

-o *filename*

Defines *filename* to be the name of the final executable program. This option overrides the default name **a.out.**

-l*library*

Searches *library* for unresolved references to functions. *library* must be an object file library in **ranlib** format.

-O Invokes the an object code optimizer.

-S Creates an assembly source listing of the compiled C source file and copies this listing to the file whose base name is the same as the source but whose extension is **.s.** This file is suitable for assembly with **as.**

-L Creates an assembler file containing assembled code and assembly source instructions. The listing is copied to the file whose base name is the same as the source but whose extension is **.L.** This option suppresses the **-S** option.

-NM *name*

Sets the module name for each compiled or assembled source file to *name*. If not given, the file name of each source file is used.

-NT *name*

Sets the text segment name for each compiled or assembled source file to *name*. If not given, the name **module_TEXT** is used for middle model and **_TEXT** is used for small model.

-ND *name*

Sets the data segment name for each compiled or assembled source file to *name*. If not given, the name **_DATA** is used.

-**NGT** *name*
Sets the text group name for each compiled or assembled source file to *name*. If not given, the name **IGROUP** is used.

-**NGD** *name*
Sets the data group name for each compiled or assembled source file to *name*. If not given, the name **DGROUP** is used.

Many options (or equivalent forms of these options) are passed to the link editor as the last phase of compilation. The **s, m,** and **l** configuration options are passed to specify memory requirements. The **-i, -F,** and **-p** options are passed to specify other characteristics of the final program.

The **-D** and **-I** options may be used several times on the command line. The **-D** option must not define the same name twice. The options affect subsequent source files only.

Memory Models

cc can create programs for three different memory models: small, middle, and large. In addition, small model programs can be pure or impure.

Impure Text Small Model
These programs occupy one segment of up to 64K bytes in which both text and data are combined. **cc** creates impure small model programs by default. Such programs can also be created with the **-Ms** option described earlier in this entry.

Pure Text Small Model
These programs occupy two segments of up to 64K bytes each. Text and data are in separate segments. The text is read-only and may be shared by several processes at once. The maximum program size is 128K bytes. Pure small model programs are created by using the **-i** and **-Ms** options.

Middle Model
These programs occupy several physical segments, but only one segment contains data. Text is divided among as many segments as required. Special calls and returns are used to access functions in other data segments. Text can be any size. Data must not exceed 64K bytes. Middle model programs are created by using the **-Mm** option. These programs are always pure.

Large Model
These programs occupy several physical segments with both text and data in as many segments as required. Special addresses are used to access data in other data segments. Text and data may be any size, but no data item can be larger than 64K bytes. Large model programs are created by using the **-Ml** option. These programs are always pure.

Small, middle, and large model object files can be linked only with object and library files of the same model. It is not possible to combine small, medium, and large object files in one executable program. **cc** occasionally selects the correct small, middle, or large versions of the standard libraries, basing its choice on the configuration option. It is up to the user to make sure that all object files and private libraries are properly compiled in the appropriate model.

The special calls and returns used in middle and large model programs may affect execution time. In particular, the execution time of a program that makes heavy use of functions and function pointers may differ noticeably from small model programs.

In both middle and large model programs, function pointers are 32 bits long. In large model programs, data pointers are 32 bits long. Programs making use of such pointers must be written carefully to avoid incorrect declaration and use of these variables. **lint** will help check for correct use.

The **-NM**, **-NT**, **-ND**, **-NGT**, and **-NGD** options may be used with middle and large model programs to direct the text and data of specific object files to named physical segments. All text having the same text segment name is placed in a single physical segment.

Files

/bin/cc

See Also

ar, as, ld, lint, ranlib

Notes

Error messages are produced by the program that detects the error. These messages are usually produced by the C compiler, but may occasionally be produced by the assembler or link loader.

All object module libraries must have a current **ranlib** directory.

cdc - Changes the delta commentary of an SCCS delta.

Syntax

cdc -r*SID* [-m[*mrlist*]] [-y[*comment*]] files

Description

cdc changes the delta commentary for the SID specified by the **-r** option, of each named SCCS file.

The delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the **delta** command (**-m** and **-y** options).

If a directory is named, **cdc** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of - (dash) is given, the standard input is read (see the "Warning" section later in this entry); each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to **cdc**, which may appear in any order, consist of option arguments, and file names.

All the described option arguments apply independently to each named file:

-r *SID*
Used to specify the SCCS identification (SID) string of a delta for which the delta commentary is to be changed.

-m[*mrlist***]**
If the SCCS file has the **v** flag set (see the entry for **admin** earlier in this appendix), then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the **-r** option may be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of **delta.** In order to delete an MR, place the character **!** ahead of the MR number. (See the "Examples" section later in this entry.) If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If **-m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read. If the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt. (See the description of the **-y** option, which follows.)

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the **v** flag has a value (see **admin**), it is taken to be the name of a program or shell procedure that validates the correctness of the MR numbers. If a nonzero exit status is returned from the MR number validation program, **cdc** terminates, and the delta commentary remains unchanged.

-y[*comment*]
Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **-r** option. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If **-y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the *comment* text.

In general, if you have made the delta, you can change its delta commentary; if you own the file and directory, you can modify the delta commentary.

Examples

The following command

```
cdc - r1.6 - m"bl78-12345 !bl77-54321 bl79-00001" - ytrouble
s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of **s.file.** The following interactive sequence has the same effect:

```
cdc -r1.6 s.file
MRs? !bl77-54321 b178-12345 bl79-00001
comments? trouble
```

Warning

If SCCS file names are supplied to the **cdc** command via the standard input (-on the command line), then the **-m** and **-y** options must also be used.

Files

x-file See **delta.**

z-file See **delta.**

See Also

admin, delta, get, help, prs

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Use **help** for explanations.

comb - Combines SCCS deltas.

Syntax

comb [-o] [-s] [-p*sid*] [-c*list*] *file* ...

Description

comb provides the means to combine one or more deltas in an SCCS file and make a single new delta. The new delta replaces the previous deltas, making the SCCS file smaller than the original.

comb does not perform the combination itself. Instead, it generates a shell procedure that you must save and execute to reconstruct the given SCCS files. **comb** copies the generated shell procedure to the standard output. To save the procedure, you must redirect the output to a file. The saved file can then be executed like any other shell procedure. (See **sh** in "Commands" in the *XENIX 286 Reference Manual*.)

When invoking **comb**, arguments may be specified in any order. All options apply to all named SCCS files. If a directory is named, **comb** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of whose path names does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

The options are as follows. Each is explained as though only one named file is to be processed, but the effects of any option apply independently to each named file.

-p*SID*
: The SCCS identification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-c*list*
: A *list* of deltas to be preserved. (For the syntax of a *list*, see the entry for **get** later in this appendix.) All other deltas are discarded.

-o
: For each **get**-**e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created; otherwise, the reconstructed file would be accessed at the most recent ancestor. Use of the **-o** option may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-s This argument causes **comb** to generate a shell procedure that will produce a report for each file giving the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by

100 * (original - combined) / original

Before any SCCS files are actually combined, you should use this option to determine exactly how much space is saved by the combining process.

If no option arguments are specified, **comb** will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

Files

comb????? Temporary files

See Also

admin, delta, get, help, prs

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Use **help** for explanations.

Notes

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to be larger than the original.

config - How to configure a XENIX system.

Syntax

/etc/config [-t] [-l *file*] [-c *file*] [-m *file*] *dfile*

Description

config is a program that takes a description of a XENIX system and generates a file that is a C program defining the configuration tables for the various devices on the system. **config** is normally invoked in the configuration process via a makefile, and not directly by the user doing configuration.

The **-c** option specifies the name of the configuration table file; **c.c** is the default name.

The **-m** option specifies the name if the file that contains all the information regarding supported devices; **/sys/conf/master** is the default name. This file is supplied with the XENIX system and should not be modified unless the user fully understands its construction.

The **-t** option requests a short table of major device numbers for character and block type devices. This can facilitate the creation of special files.

The caller must supply *dfile* (normally **/sys/conf/xenixconf**); it must contain device information for the user's system. This file is divided into two parts. The first part contains physical device specifications. The second part contains system-dependent information. Any line with an asterisk (*) in column 1 is a comment.

All configurations are assumed to have a set of required devices, which must be present to run XENIX, such as the system clock. These devices must not be specified in *dfile*.

First Part of *dfile*

Each line contains two fields delimited by blanks and/or tabs in the following format:

devname number

where *devname* is the name of a device as it appears in the **/sys/conf/master** device table, and *number* is the decimal number of devices associated with the corresponding controller. *number* is optional. If omitted, the default value is used, which is the maximum value for that controller.

Certain drivers that may be provided with the system are actually pseudo-device drivers. That is, no real hardware is associated with them.

Second Part of *dfile*

The second part contains three different types of lines. Note that all specifications are required, although their order is arbitrary.

1. Root/pipe device specification

Each line has three fields:

root	*devname*	*minor*
pipe	*devname*	*minor*

where *minor* is the minor device number (in octal).

2. Swap device specification

This single line contain five fields, as follows:

swap	*devname*	*minor*	*swplo*	*nswap*

where swplo is the lowest disk block (decimal) in the swap area and nswap is the number of disk blocks (decimal) in the swap area.

3. Parameter specification

A number of lines of two fields each, as follows:

buffers	*number*
inodes	*number*
files	*number*
mounts	*number*
swapmap	*number*
pages	*number*
calls	*number*
procs	*number*
maxproc	*number*
texts	*number*
clists	*number*
locks	*number*
timezone	*number*
daylight	0 or 1

Example

Suppose you wanted to configure a system with the following devices:

one hard disk controller (**hd**) with one device
one flexible disk controller (**fd**) with one device

You must also specify the following parameter information:

root device is **hd** (pseudo disk 3)
pipe device is **hd** (pseudo disk 3)
swap device is **hd** (pseudo disk 2)
with a *swplo* of 1 and an *nswap* of 2300
number of buffers is 50
number of processes is 50
maximum number of processes per user ID is 15
number of mounts is 8
number of inodes is 120
number of files is 120
number of calls is 30
number of texts is 35
number of character buffers is 150
number of swapmap entries is 50
number of memory pages is 512
number of file locks is 100
timezone is pacific time
daylight time is in effect

The actual system configuration would be specified as follows:

```
hd                      1
fd                      1
root        hd          3
pipe        hd          3
swap        hd          2           0           2300

* Comments may be inserted in this manner.

buffers                 50
procs                               150
maxproc                 15
mounts                  8
inodes                  120
files                               120
calls                               30
texts                               35
clists                              150
swapmap                 50
pages                               (1024/2);
locks                               100
timezone                (8*60)
daylight                1
```

Files

/sys/conf/master	Default master device table
c.c	Default output configuration table file

See Also

master in "File Formats" in the *XENIX 286 C Library Guide*

XENIX 286 Installation and Configuration Guide

XENIX 286 Device Driver Guide

Diagnostics

Diagnostics are routed to the standard output and are self-explanatory.

Notes

The **-t** option does not know about devices that have aliases. However, the major device numbers are always correct.

cref - Makes a cross-reference listing.

Syntax

cref [-acilnostux123] *file* ...

Description

cref makes a cross-reference listing of assembler or C programs. The program searches the given files for symbols in the appropriate C or assembly language syntax.

The output report is in four columns:

1. Symbol
2. File name
3. Current symbol or line number
4. Text as it appears in the file

cref uses either an **ignore** file or an **only** file. If the **-i** option is given, the next argument is taken to be an **ignore** file; if the **-o** option is given, the next argument is taken to be an **only** file. **ignore** and **only** files are lists of symbols separated by newlines. All symbols in an **ignore** file are ignored in columns 1 and 3 of the output. If an **only** file is given, only symbols in that file will appear in column 1. Only one of these options may be given; the default setting is **-i** using the default **ignore** file. (See the "Files" section later in this entry.) Assembler-predefined symbols and C keywords are ignored.

The **-s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, it is the current function name. The **-l** option causes the line number within the file to be put in column 3.

The **-t** option causes the next available argument to be used as the name of the intermediate file (instead of the temporary file **/tmp/crt??**). This file is created and is not removed at the end of the process.

The **cref** options are

a Uses assembler format (default)

c Uses C format

i Uses an **ignore** file

l Puts line number (instead of current symbol) in column 3

n Omits column 4 (no context)

o Uses an **only** file (see above)

s Current symbol in column 3 (default)

t User-supplied temporary file

u Prints symbols that occur exactly once

x Prints C-external symbols

1 Sorts output on column 1 (default)

2 Sorts output on column 2

3 Sorts output on column 3

Files

/usr/lib/cref/* Assembler-specific files

See Also

as, cc, xref

sort in "Commands" in the *XENIX 286 Reference Manual*

Notes

cref inserts an ASCII DEL character into the intermediate file after the eighth character of each name that is eight or more characters long in the source file.

csh - Invokes a shell command interpreter with C-like syntax.

Syntax

csh [-cefinstvVxX] [*arg*] ...

Description

csh is a command language interpreter. It begins by executing commands from the file **.cshrc** in the home directory of the invoker. If this is a login shell, then it also executes commands from the file **.login** there. In the normal case, the shell will then begin reading commands from the terminal, prompting with **%**. (The processing of arguments and the use of the shell to process files containing command scripts is described later.)

The shell then repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and then parsed. Finally, each command in the current line is executed.

When a login shell terminates, it executes commands from the file **.logout** in the user's home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (,) form separate words. If doubled in &&, ||, <<, or >>, these pairs form single words. These parser metacharacters may be made part of other words or denied their special meaning by placing a backslash (\) ahead of them. A newline preceded by a \ is equivalent to a blank.

In addition, strings enclosed in matched pairs of quotations--', `, or "--form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. The semantics for these quotations will be described later in this entry. Within pairs of \ or " characters, a newline preceded by a \ gives a true newline character.

When the shell's input is not a terminal, the character # introduces a comment that continues to the end of the input line. The character does not have this special meaning when preceded by \ and placed inside the quotation marks ', `, and ".

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by vertical bar (|) characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ; and executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an ampersand (&). Such a sequence cannot be terminated by a hangup signal; the **nohup** command need not be used.

Any of these characters may be placed in parentheses to form a simple command, which may be a component of a pipeline. It is also possible to separate pipelines with || or && indicating, as in the C language, that the second is to be executed only if the first fails or succeeds, respectively. (See the section on "Expressions" later in this entry.)

Substitutions

The following sections describe the various transformations the shell performs on the input. It performs substitutions in the order the input demands, not necessarily in the order presented here.

History Substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus, history substitutions provide a generalization of a **redo** function.

History substitutions begin with the exclamation point character (!) and may begin anywhere in the input stream if a history substitution is not already in progress. This ! may be preceded by a \ to prevent its special meaning. An ! is passed unchanged when it is followed by a blank, tab, newline, =, or (. History substitutions also occur when an input line begins with a caret (^). This special abbreviation will be described later.

Any input line that contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal that consist of one or more words are saved on the history list, the size of which is controlled by the **history** variable. The previous command is always retained. Commands are numbered sequentially from 1.

For example, consider the following output from the history command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an ! in the prompt string.

With the current event 13, we can refer to previous events by event number **!11**. We can also refer to them relatively, as in **!-2** (which refers to the same event), by a prefix of a command word, as in **!d** for event 12 or **!w** for event 9, or by a string contained in a word in the command, as in **!?mic?** (which also refers to event 9). These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case, **!!** refers to the previous command; thus **!!** alone is essentially a **redo**. The form **!#** references the current command (the one being typed in). It allows a word to be selected from further left in the line to avoid retyping a long name, as in **!#:1**.

To select words from an event, we can follow the event specification by a colon (:) and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, and so on. The basic word designators are

0	First (command) word
n	*n*th argument
^	First argument (that is, 1)
$	Last argument
%	Word matched by (immediately preceding) **?s?** search
x-*y*	Range of words
-*y*	Abbreviates **0**-*y*
*	Abbreviates **^-$**, or nothing if only one word in event
*x**	Abbreviates **x-$**
x-	Like *x** but omitting word **$**

The : separating the event specification from the word designator can be omitted if the argument selector begins with a ^, $, *, -, or %. A sequence of modifiers, each preceded by a colon, can be placed after the optional word designator. The following modifiers are defined:

h	Removes a trailing path name component
r	Removes a trailing **.xxx** component
s/*l*/*r*/	Substitutes *l* for *r*
t	Removes all leading path name components
&	Repeats the previous substitution
g	Applies the change globally, prefixing the above
p	Prints the new command but do not execute it
q	Quotes the substituted words, preventing substitutions
x	Like **q**, but breaks into words at blanks, tabs, and newlines

The modification is applied only to the first modifiable word unless preceded by a **g**. In any case, it is an error for no word to be applicable.

The left sides of substitutions are strings, not regular expressions in the sense of the editors. Any character can be used in place of slash (/) as the delimiter. A backslash (\) quotes the delimiter into the **l** and **r** strings. An & in the right side is replaced by the text from the left. A \ quotes &. A null *l* uses the previous string either from an *l* or from a contextual scan string **s** in **!?s?**. The trailing delimiter in a substitution or the trailing ? in a contextual scan can be omitted if a newline follows immediately.

A history reference may be given without an event specification, for example, **!$**. In this case, the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus **!?foo?^!$** gives the first and last arguments from the command matching **?foo?**.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a caret (^). This is equivalent to **!:s^** and provides a convenient shorthand for substitutions on the text of the previous line. Thus, **^lb^lib** fixes the spelling of **lib** in the previous command. Finally, a history substitution may be surrounded with braces ({ }) if necessary to insulate it from the characters that follow. Thus, after **ls -ld~paul** we might do **!{l}a** to do **ls -ld~paula**, while **!la** would look for a command starting **la**.

Quotations with ' and "

The quotation of strings by ' and " can be used to prevent all or some of the remaining substitutions. Strings enclosed in ' are prevented any further interpretation. Strings enclosed in " are variable, and command expansion may occur.

In both cases, the resulting text becomes (all or part of) a single word. Only in one special case (see "Command Substitution" later in this entry) does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias Substitution

The shell maintains a list of aliases that can be established, displayed, and modified by the **alias** and **unalias** commands. After a command line is scanned, it is parsed into distinct commands, and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text that is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for **ls** is **ls -l** the command **"ls /usr"** would map to **"ls -l /usr"**. Similarly, if the alias for **lookup** was **grep "!^ /etc/passwd"** then **"lookup bill"** would map to **"grep bill /etc/passwd"**.

If an alias is found, the word transformation of the input text is performed, and the aliasing process begins again on the reformed input line. If the first word of the new text is the same as the old, flagging it will prevent further aliasing and will preclude looping. Other loops are detected and cause an error.

The mechanism allows aliases to introduce parser metasyntax. Thus, we can alias **print** as **'"pr\!* | lpr"'** to make a command that paginates its arguments to the line printer.

Variable Substitution

The shell maintains a set of variables, each of which has as its value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the **argv** variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the **set** and **unset** commands. Of the variables referred to by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the **verbose** variable is a toggle which causes command input to be echoed. The setting of this variable results from the **-v** command line option.

Other operations treat variables numerically. The at sign (**@**) command permits numeric calculations to be performed and the result assigned to a variable. However, variable values are always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed, keyed by dollar sign ($) characters. This expansion can be prevented by placing a backslash (\) ahead of the dollar sign except within double quotation marks (") where it always occurs and within single quotation marks (') where it never occurs. Strings quoted by back quotation marks (`) are interpreted later (see the section "Command Substitution" later in this entry) so dollar sign substitution does not occur there until later, if at all. A dollar sign is passed unchanged if followed by a blank, tab, or end-of-line.

Input and output redirections are recognized before variable expansion and are variable-expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to generate more than one word, the first of which becomes the command name and the rest of which become arguments.

Unless enclosed in double quotation marks or given the **:q** modifier, the results of variable substitution may eventually be command- and file-name-substituted. Within double quotation marks (") a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variable's value separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands to multiple words with each word separated by a blank and quoted to prevent later command or file name substitution.

The following sequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable that is not set.

$*name*
${*name***}**
Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters, which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If *name* is not a shell variable, but is set in the environment, then that value is returned. However, modifiers and the other forms given below are not available in this case.

$*name***[***selector***]**
${*name***[***selector***] }**
May be used to select only some of the words from the value of *name*. The selector is subjected to $ substitution and may consist of a single number or two numbers separated by a -. The first word of a variable's value is numbered 1. If the first number of a range is omitted, it defaults to 1. If the last member of a range is omitted, it defaults to **$**~*ame*. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$~*ame*
${~*ame***}**
Gives the number of words in the variable. This is useful for later use in a [selector].

$0 Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$*number*
${*number***}**
Equivalent to **$argv[***number***]**.

$*** Equivalent to **$argv[*].

The modifiers **:h, :t, :r, :q** and **:x** may be applied to these substitutions as may **:gh, :gt** and **:gr**. If braces ({ }) appear in the command form, then the modifiers must appear within the braces. Only one colon (:) modifier is allowed on each dollar sign ($) expansion.

The following substitutions may not be modified with : modifiers.

$?*name*
${?*name***}**
Substitutes the string 1 if *name* is set, 0 if it is not.

$?0 Substitutes 1 if the current input file name is known, 0 if it is not.

$$ Substitutes the (decimal) process number of the (parent) shell.

Command and File Name Substitution

Command and file name substitution are applied selectively to the arguments of built-in commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands that are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command Substitution. Command substitution is indicated by a command enclosed in back quotation marks (`). The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within double quotation marks, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

File Name Substitution. If a word contains any of the characters *, ?, [, or { or begins with the character ~, then that word is a candidate for file name substitution, also known as globbing. This word is then regarded as a pattern and replaced with an alphabetically sorted list of file names that match the pattern. In a list of words specifying file name substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching, the characters ~ and { being more akin to abbreviations.

In matching file names, the character dot (.) at the beginning of a file name or immediately following a slash (/), as well as the character / itself must be matched explicitly. The character asterisk (*) matches any string of characters, including the null string. The character question mark (?) matches any single character. The sequence [...] matches any one of the characters enclosed. Within [...], a pair of characters separated by dash (-) matches any character lexically between the two (inclusive).

The character tilde (~) at the beginning of a file name is used to refer to home directories. Standing alone, it expands to the invoker's home directory as reflected in the value of the variable **home.** When followed by a name consisting of letters, digits and dash characters (-), the shell searches for a user with that name and substitutes his or her home directory; thus **~ken** might expand to **/usr/ken** and **~ken/chmach** to **/usr/ken/chmach.** If the character ~ is followed by a character other than a letter or slash (/) or if it appears somewhere besides the beginning of a word, it is left unchanged.

The metanotation **a{b,c,d}e** is a shorthand for **abe ace ade.** Left-to-right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus, **~source/s1/{oldls,ls}.c** expands to **/usr/source/s1/oldls.c /usr/source/s1/ls.c,** whether or not these files exist and without any chance of error if the home directory for source is **/usr/source.** Similarly, **../{memo,*box}** might expand to **../memo ../box ../mbox.** (Note that **memo** was not sorted with the results of matching ***box.**) As a special case **{, }** and **{}** are passed unchanged.

Input/Output

The standard input and standard output of a command may be redirected with the following syntax:

<name

Opens file *name* (which is first variable, command, and file name expanded) as the standard input.

<<word

Reads the shell input up to a line which is identical to *word*. *word* is not subjected to variable, file name or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting backslash, double or single quotation mark, or back quotation mark appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \, and `. Commands that are substituted have all blanks, tabs, and newlines preserved, except for the final newline, which is dropped. The resulting text is placed in an anonymous temporary file that is given to the command as standard input.

>name
>!name
>&name
>&!name

The file *name* is used as standard output. If the file does not exist, then it is created; if the file exists, it is truncated, and its previous contents are lost.

If the variable **noclobber** is set, then the file must not already exist, or it must be a character special file (for example, a terminal or **/dev/null**); otherwise, an error results. This provision helps prevent accidental destruction of files. In this case the ! forms can be used to suppress a check. The forms involving & route the diagnostic output into the specified file as well as the standard output. *name* is expanded in the same way as < input file names are.

>>name
>>&name
>>!name
>>&!name

Uses the file *name* as standard output like > but places output at the end of the file. If the variable **noclobber** is set, then it is an error for the file not to exist unless one of the ! forms is given. Otherwise similar to >.

If a command is run detached (followed by &), then the default standard input for the command is the empty file **/dev/null**. Otherwise, the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather, they receive the original standard input of the shell. The << mechanism should be used to present in-line data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form **|&** instead of just **|**.

Expressions

A number of the built-in commands (to be described later) take expressions, in which the operators are similar to those of C and have the same precedence. These expressions appear in the **@, exit, if,** and **while** commands. The following operators are available:

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

Here, the precedence increases from left to right, with the operators

```
==  and  !=
<=, >=, <, and >
<< and >>
+ and -
*/ and %
```

forming groups at the same level. The == and **!**= operators compare their arguments as strings; all others operate on numbers. Strings that begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The results of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word. Except when adjacent to components of expressions that are syntactically significant to the parser--& | < > ( )--they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in { and } and file enquiries of the form *-l name* where *l* is one of

r Read access
w Write access
x Execute access
e Existence
o Ownership
z Zero size
f Plain file
d Directory

The specified *name* is command- and file-name-expanded, then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all enquiries return false (0). Command executions succeed, returning true (1) if the command exits with status 0; otherwise, they fail, returning false (0). If more detailed status information is required, then the command should be executed outside of an expression and the variable **status** examined.

Control Flow

The shell contains a number of commands that can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The **foreach, switch,** and **while** statements, as well as the **if-then-else** form of the **if** statement require that the major keywords appear in a single simple command on an input line as shown in this section.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto commands will succeed on nonseekable inputs.)

Built-In Commands

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, then it is executed in a subshell.

alias
alias *name*
alias *name wordlist*

The first form prints all aliases. The second form prints the alias for *name*. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command- and file-name-substituted. *name* is not allowed to be **alias** or **unalias.**

break

Causes execution to resume after the **end** of the nearest enclosing **foreach** or **while** statement. The remaining commands on the current line are executed. Multilevel breaks are thus possible by writing them all on one line.

breaksw

Causes a break from a **switch,** resuming after the **endsw.**

case label:

A label in a **switch** statement as discussed later.

cd
cd *name*
chdir
chdir *name*

Changes the shell's working directory to directory *name*. If no argument is given, then the directory changes to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with **/**, **./**, or **../**), then each component of the variable **cdpath** is checked to see if it has a subdirectory **name.** Finally, if all else fails but *name* is a shell variable whose value begins with **/**, then this is tried to see if it is a directory.

continue
Continues execution of the nearest enclosing **while** or **foreach.** The rest of the commands on the current line are executed.

default:
Labels the default case in a **switch** statement. The default should come after all **case** labels.

echo *wordlist*
The specified words are written to the shell's standard output. A **\c** causes the echo to complete without printing a newline. A **\n** in *wordlist* causes a newline to be printed. Otherwise, the words are echoed, separated by spaces.

else
end
endif
endsw
See the description of the **foreach, if, switch,** and **while** statements, which follows.

exec *command*
The specified *command* is executed in place of the current shell.

exit
exit(*expr***)**
The shell exits either with the value of the **status** variable (first form) or with the value of the specified *expr* (second form).

foreach *name* **(***wordlist***)**
...
end
The variable *name* is successively set to each member of *wordlist,* and the sequence of commands between this command and the matching **end** is executed. (Both **foreach** and **end** must appear alone on separate lines.)

The built-in command **continue** may be used to continue the loop prematurely and the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read up once, prompting with ?, before any statements in the loop are executed.

glob *wordlist*
Like **echo** except that \ escapes are not recognized and that words are delimited by null characters in the output. Useful for programs that need the shell to file-name-expand a list of words.

goto *word*
The specified *word* is file-name- and command-expanded to yield a string of the form *label.* The shell rewinds its input as much as possible and searches for a line of the form *label***:** possibly preceded by blanks or tabs. Execution continues after the specified line.

history
Displays the history event list.

if (*expr***)** *command*

If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the **if** command. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. When *command* is not executed, input/output redirection occurs even if *expr* is false.

if (*expr***) then**
...
else if (*expr2***) then**
...
else
...
endif

If the specified *expr* is true, then the commands to the first **else** are executed; else if *expr2* is true, then the commands to the second else are executed, etc. Any number of **else-if** pairs are possible; only one **endif** is needed. The **else** part is likewise optional. (The words **else** and **endif** must appear at the beginning of input lines; the **if** must appear alone on its input line or after **$IR else.**)

logout

Terminates a login shell. The only way to log out if **ignoreeof** is set.

nice
nice *+number*
nice *command*
nice *+number command*

The first form sets the **nice** for this shell to 4. The second form sets the **nice** to the given *number*. The final two forms run command at priority 4 and *number*, respectively. With **"nice** *-number* **...."** the super-user may increase the priority by reducing the priority number. The command is always executed in a subshell, and the restrictions placed on commands in simple **if** statements apply.

nohup
nohup *command*

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified *command* to be run with hangups ignored. Unless the shell is running detached, **nohup** has no effect. All processes detached with & are automatically **nohup**ed. (Thus, **nohup** is not really needed.)

onintr
onintr-
onintr *label*

Controls the action of the shell on interrupts. The first form restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. The second form **onintr-** causes all interrupts to be ignored. The final form causes the shell to execute a **goto** *label* when an interrupt is received or a child process terminates because it was interrupted. In any case, if the shell is running detached and interrupts are being ignored, no forms of **onintr** have meaning, and interrupts continue to be ignored by the shell and all invoked commands.

rehash

Causes the internal hash table of the contents of the directories in the **path** variable to be recomputed. This is needed if new commands are added to directories in the **path** while you are logged in. This should be necessary only if you add commands to one of your own directories or if a systems programmer changes the contents of one of the system directories.

repeat *count command*

The specified *command*, which is subject to the same restrictions as the *command* in this one-line **if** statement, is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

set
set *name*
set *name=word*
set *name*[*index*]=*word*
set *name*=(*wordlist*)

The first form of the command shows the value of all shell variables. Variables that have a value other than a single word print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index*th component of *name* to word; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases, the value is command- and file-name-expanded. These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv *name value*

Sets the value of the environment variable *name* to be *value*, a single string. Useful environment variables are **TERM**, the type of terminal you are using, and **SHELL**, the shell you are using.

shift
shift *variable*

The members of **argv** are shifted to the left, discarding **argv[1]**. It is an error for **argv** not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source *name*

The shell reads commands from *name*. **source** commands may be nested. If they are nested too deeply, the shell may run out of file descriptors. An error in a **source** at any level terminates all nested **source** commands. Input during **source** commands is never placed on the history list.

switch (*string*)
case *str1*:
...
breaksw
...
default:
...
breaksw
endsw

Each case label is successively matched against the specified *string*, which is first command- and file-name-expanded. The file metacharacters *, ?, and [...] may be used in the case labels, which are variable-expanded. If none of the labels matches before a default label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command **breaksw** causes execution to continue after the **endsw.** Otherwise, control may fall through case labels and default labels, as in C. If no label matches and there is no default, execution continues after the **endsw.**

time
time *command*

With no argument, a summary of time used by this shell and its children is printed. If arguments are given, the specified simple command is timed, and a time summary as described under the **time** variable (see "Predefined Variables" later in this entry) is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask *value*

The file creation mask is displayed (first form) or set to the specified *value* (second form). The mask is given in octal. Common values for the mask are 002, which gives all access to the group and read and execute access to others, or 022, which gives all access except write access to users in the group and to others.

unalias *pattern*

All aliases with names that match the specified *pattern* are discarded. Thus, all aliases are removed by **unalias *.** It is not an error for nothing to be **unaliased.**

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unset *pattern*

All variables with names that match the specified *pattern* are removed. Thus all variables are removed by **unset *;** this has noticeably distasteful side effects. Having nothing **unset** is not an error.

unsetenv *pattern*

All environment variables with names that match the specified *pattern* are removed.

wait

All child processes are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (*expr*)
...
end

As the specified expression evaluates nonzero, the commands between the **while** and the matching **end** are evaluated. **break** and **continue** may be used to terminate or continue the loop prematurely. (The **while** and **end** must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the **foreach** statement if the input is a terminal.

@
@ *name = expr*
@ *name*[*index*] = *expr*

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains <, >, &, or |, then at least this part of the expression must be placed within ( ). The third form assigns the value of *expr* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

Assignment operators such as *= and += are available as in C. The space separating the name from the assignment operator is optional. Spaces are mandatory in separating components of *expr* that would otherwise be single words. Special postfix operators ++ and --, respectively, increment and decrement *name* (for example, **@ i++**).

Predefined Variables

The following variables have special meaning to the shell. Of these, **argv, child, home, path, prompt, shell,** and **status** are always set by the shell. Except for **child** and **status,** setting occurs only at initialization; these two variables will not then be modified unless done explicitly by the user.

The shell copies the environment variable **PATH** into the variable **path** and copies the value back into the environment whenever **path** is set. Thus, there is no need to be concerned about its setting other than in the file **.cshrc,** as inferior **csh** processes will import the definition of **path** from the environment.

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted; that is, **$1** is replaced by **$argv[1],** etc.

cdpath Gives a list of alternate directories searched to find subdirectories in **cd** commands.

child The process number printed when the last command was forked with &. This variable is **unset** when this process terminates.

echo Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For nonbuilt-in commands, all expansions occur before echoing. Built-in commands are echoed before command and file name substitution, since these substitutions are then done selectively.

histchars Can be assigned a two-character string. The first character is used as a history character in place of !; the second character is used in place of the ^ substitution mechanism. For example, **set histchars=",;"** will cause the history characters to be comma and semicolon.

history Can be given a numeric value to control the size of the history list. Any command that has been referenced in this many events will not be discarded. A **history** that is too large may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The file name expansion of ~ refers to this variable.

ignoreeof If set, the shell ignores end-of-file from input devices that are terminals. This prevents a shell from accidentally being terminated by typing a CONTROL-D.

mail The files where the shell checks for mail. This is done after each command completion and will result in a prompt, if a specified interval has elapsed. The shell says **You have new mail** if the file exists with an access time not greater than its modify time. If the first word of the value of **mail** is numeric, it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes. If multiple mail files are specified, then the shell says **New mail in name** when there is mail in the file **name.**

noclobber As described earlier, in the "Input/Output" section of this entry, restrictions are placed on output redirection to ensure that files are not accidentally destroyed and that >> redirections refer to existing files.

noglob If set, file name expansion is inhibited. This is most useful in shell scripts that are not dealing with file names, and when a list of file names has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a file name expansion to not match any existing files; rather, the primitive pattern is returned. It is still an error for the primitive pattern to be malformed; that is, **echo [** still gives an error.

path Each word of the **path** variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no **path** variable, then only full path names will execute. The usual search path is **/bin, /usr/bin,** and **.**, but this may vary from system to system. For the super-user, the default search path is **/etc, /bin,** and **/usr/bin.** A shell that is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the **path** variable after reading **.cshrc** and each time the **path** variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the **rehash.** Otherwise, the commands may not be found.

prompt The string printed before each command is read from an interactive terminal input. If an ! appears in the string, it will be replaced by the current event number unless a preceding \ is given. Default is % for the user, # for the super-user.

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set but are not executable by the system. (See the section "Nonbuilt-In Command Execution," which follows.) Initialized to the system-dependent home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Abnormal termination results in a core dump. Built-in commands that fail return exit status 1; all other built-in commands set status 0.

time Controls automatic timing of commands. If set, then any command that takes more than this many CPU seconds will cause a line of information to be printed when the command terminates. This line gives the user, system, and real times and a utilization percentage, which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Nonbuilt-In Command Execution

When a command to be executed is found to not be a built-in command the shell attempts to execute the command via **exec**. Each word in the variable **path** names a directory from which the shell will attempt to execute the command. If it is given neither a **-c** option nor a **-t** option, the shell will hash the names in these directories into an internal table so that it will try an **exec** in a directory only if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via **unhash**) or if the shell was given a **-c** or **-t** argument and in any case for each directory component of **path** which does not begin with a /, the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus, **(cd ; pwd) ; pwd** prints the **home** directory, leaving you where you were (printing this after the home directory), while **cd ; pwd** leaves you in the home directory. Parenthesized commands are most often used to prevent **cd** from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands, and a new shell is spawned to read it.

If there is an **alias** for **shell** then the words of the alias will be prepended to the argument list to form the shell command. The first word of the **alias** should be the full path name of the shell (for example, **$shell**). Note that this is a special, late occurring case of **alias** substitution and that it only allows words to be prepended to the argument list without modification.

Argument List Processing

If argument 0 to the shell is -, then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument, which must be present. Any remaining arguments are placed in **argv.**

-e The shell exits if any invoked command terminates abnormally or yields a nonzero exit status.

-f The shell will start faster because it will neither search for nor execute commands from the file **.cshrc** in the invoker's home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears not to be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A \ may be used to escape the newline at the end of this line and continue on to another line.

-v Causes the **verbose** variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the **echo** variable to be set, so that commands are echoed immediately before execution.

-V Causes the **verbose** variable to be set even before **.cshrc** is executed.

-X Causes the **echo** variable to be set even before **.cshrc** is executed.

If arguments remain after flag arguments are processed but none of the **-c, -i, -s,** or **-t** options were given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file and saves its name for possible resubstitution by **$0.** Since most shell scripts on a typical system are written for the standard shell **sh,** the C shell will execute such a standard shell if the script does not start with a comment (the first character of a script is not a #). (See the entry **sh** in "Commands" in the *XENIX 286 Reference Manual.*) Remaining arguments initialize the variable **argv.**

Signal Handling

The shell normally ignores **quit** signals. The **interrupt** and **quit** signals are ignored for an invoked command if the command is followed by &; otherwise, the signals have the values the shell inherited from its parent. The shell's handling of interrupts can be controlled by **onintr.** Login shells catch the **terminate** signal; otherwise, this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the **.logout** file.

Files

~/.cshrc	Read by each shell at the beginning of execution
~/.login	Read by login shell after **.cshrc** at login
~/.logout	Read by login shell at logout
/bin/sh	Shell for scripts not starting with a **#**
/tmp/sh*	Temporary file for **<<**
/dev/null	Source of empty file
/etc/passwd	Source of home directories for **~name**

Limitations

Words can be no longer than 512 characters. The number of arguments to a command that involves file name expansion is limited to one-sixth the number of characters allowed in an argument list, which is 5120, less the characters in the environment. Command substitutions may replace no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of **alias** substitutions on a single line to 20.

See Also

sh in "Commands" in the *XENIX 286 Reference Manual*

a.out, environ in "File Fromats" in the *XENIX 286 C Library Guide*

access, exec, fork, pipe, signal, umask, wait in "System Functions" in the *XENIX 286 C Library Guide*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

Built-in control structure commands like **foreach** and **while** cannot be used with |, &, or ;.

Commands within loops, prompted for by ?, are not placed in the **history** list.

It is not possible to use colon (:) modifiers on the output of command substitutions.

csh attempts to import and export the **PATH** variable for use with regular shell scripts. This works only in simple cases, where **PATH** contains no command characters.

This version of **csh** does not support or use the process control features of the 4th Berkeley Distribution.

ctags - Creates a tags file.

Syntax

ctags [-u] [-w] [-x] *name* ...

Description

ctags makes a tags file for **vi** from specified C sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, **vi** can quickly find these function definitions.

If the **-x** flag is given, **ctags** produces a list of function names, the line number and file name on which each is defined, and the text of that line, and prints this list on the standard output. This is a simple index which can be printed out as an off-line readable function index.

Files with names ending in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions.

Other options are

-w Suppresses warning diagnostics.

-u Causes the specified files to be **updated** in tags; that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way that is rather slow. It is usually faster to simply rebuild the tags file.)

The tag **main** is treated specially in C programs. The tag formed is created by prepending **M** to the name of the file. The trailing **.c**, if any, is removed and leading path name components also removed. This makes use of **ctags** practical in directories with more than one program.

Files

tags Output tags file

See Also

ex, vi in "Commands" in the *XENIX 286 Reference Manual*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

delta - Makes a delta (change) to an SCCS file.

Syntax

delta [-r*SID*] [-s] [-n] [-g*list*] [-m[*mrlist*]] [-y[*comment*]] [-p] *files*

Description

delta is used to permanently introduce into the named SCCS files changes that were made to the files retrieved by **get** (called the g-files, or generated files).

delta makes a delta to each SCCS file named by **files.** If a directory is named, **delta** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read (see the "Warning" section later in this entry); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain options specified and the flags that may be present in the SCCS file. (See **-m** and **-y** options below.)

Options apply independently to each named file.

-r *SID*

Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more versions of the same SCCS file have been retrieved for editing (**get -e**) by the same person (login name). The *SID* value specified with the **-r** keyletter can be either the SID specified on the **get** command line or the SID to be made as reported by the **get** command. (See the entry for **get** later in this appendix.) A diagnostic results if the specified SID is ambiguous or if it is necessary but omitted from the command line.

-s Suppresses the created delta's SID, as well as the number of lines inserted, deleted, and unchanged in the SCCS file from being issued to the standard output.

-n Specifies retention of the edited g-file (normally removed at completion of delta processing).

-g*list*

Specifies a *list* of deltas that are to be ignored when the file is accessed at the change level (SID) created by this delta. (See **get** for the definition of *list*.)

-m[*mrlist***]**

If the SCCS file has the **v** flag set (see the entry for **admin** later in this appendix), then a Modification Request (MR) number must be supplied as the reason for creating the new delta.

If **-m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read. If the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt. (See the description of **-y**, which follows.)

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the **v** flag has a value (see **admin**), it is taken to be the name of a program (or shell procedure) that will validate the correctness of the MR numbers. If a nonzero exit status is returned from an MR number validation program, **delta** terminates. (It assumes that the MR numbers were not all valid.)

-y[*comment***]**

Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If **-y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read. If the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

-p Causes **delta** to print (on the standard output) the SCCS file differences before and after the delta is applied. Differences are displayed in a **diff** format.

Files

All files of the form *?-file* are explained in Chapter 5, "SCCS: Source Code Control System." The naming conventions for these files are also described.

g-file Existed before the execution of **delta**; removed after completion of **delta**.

p-file Existed before the execution of **delta**; may exist after completion of **delta**.

q-file Created during the execution of **delta**; removed after completion of **delta**.

x-file Created during the execution of **delta**; renamed to SCCS file after completion of **delta**.

z-file Created during the execution of **delta**; removed during the execution of **delta**.

d-file Created during the execution of **delta**; removed after completion of **delta**.

/usr/bin/bdiff

Program to compute differences between the "retrieved" file and the g-file.

Warning

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see **sccsfile** in "File Formats" in the *XENIX 286 C Library Guide*) and will cause an error.

A **get** of many SCCS files followed by a **delta** of those files should be avoided when the **get** generates a large amount of data. Instead, multiple **get/delta** sequences should be used.

If the standard input (-) is specified on the **delta** command line, the **-m** (if necessary), and **-y** options must also be present. Omission of these options causes an error to occur.

See Also

admin, get, help, prs

bdiff, diff in "Commands" in the *XENIX 286 Reference Manual*

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Use **help** for explanations.

get - Gets a version of an SCCS file.

Syntax

get [-rSID] [-c*cutoff*] [-i*list*] [-x*list*] [-a*seq-no.*] [-k] [-e]
[-l[p]] [-p] [-m] [-n] [-s] [-b] [-g] [-t] *file* ...

Description

get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with -. The arguments may be specified in any order, but all options apply to all named SCCS files. If a directory is named, **get** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of whose path names does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the g-file. Its name is derived from the SCCS file name by simply removing the leading "s.". (See the "Files" section later in this entry.)

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any option apply independently to each named file.

-r*SID*
: The SCCS identification string (SID) of the version (delta) of an SCCS file to be retrieved.

-c*cutoff*
: *cutoff* date-time, in the form

  YY[MM[DD[HH[MM[SS]]]]]

  No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **-c7502** is equivalent to **-c750228235959.** Any number of nonnumeric characters may separate the various two-digit pieces of the *cutoff* date-time. This feature allows you to specify a *cutoff* date in the form: "-c77/2/2 9:22:25".

-e
: Indicates that the **get** is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of **delta.** The **-e** option used in a **get** for a particular version (SID) of the SCCS file prevents further **get**s for editing on the same SID until **delta** is executed or the **j** (joint edit) flag is set in the SCCS file. (See the entry for **admin** later in this appendix.) Concurrent use of **get -e** for different SIDs is always allowed.

If the g-file generated by **get** with an **-e** option is accidentally ruined in the editing process, it may be regenerated by re-executing the **get** command with the **-k** option in place of the **-e** option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see **admin**) are enforced when the **-e** option is used.

-b Used with the **-e** option to indicate that the new delta should have an SID in a new branch. This option is ignored if the **b** flag is not present in the file (see **admin**) or if the retrieved **delta** is not a leaf **delta**. (A leaf **delta** is one that has no successors on the SCCS file tree.)

Note: A branch **delta** may always be created from a nonleaf **delta**.

-i*list*

A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> :: = <range> | <list> , <range>
<range> :: = SID | SID - SID

SID, the SCCS identification of a delta, may be in any form described in [get citation].

-x*list*

A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **-i** option for the *list* format.

-k Suppresses replacement of identification keywords in the retrieved text by their value. The **-k** option is implied by the **-e** option.

-l[p]

Causes a delta summary to be written into an l-file. If **-lp** is used, then an l-file is not created; the delta summary is written on the standard output instead. See the "Files" section later in this entry for the format of the l-file.

-p Causes the text retrieved from the SCCS file to be written on the standard output. No g-file is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the **-s** option is used, in which case it disappears.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, a horizontal tab, the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value. The format is: %M% value, a horizontal tab, the text line. When both the **-m** and **-n** options are used, the format is: %M% value, a horizontal tab, the **-m** option-generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an l-file or to verify the existence of a particular SID.

-t Used to access the most recently created (top) delta in a given release (for example, **-r1**) or release and level (**-r1.2**).

-a*seq-no*.
The delta sequence number of the SCCS file delta (version) to be retrieved. (See **sccsfile** in "File Formats" in *XENIX 286 C Library Guide*.) This option is used by the **comb** command; it is not particularly useful and should be avoided. If both the **-r** and **-a** options are specified, the **-a** option is used. Care should be taken when using the **-a** option in conjunction with the **-e** option, as the SID of the delta to be created may not be what you expect. The **-r** option can be used with the **-a** and **-e** options to control the naming of the SID of the delta to be created.

For each file processed, **get** responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **-e** option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a newline) before it is processed. If the **-i** option is used, included deltas are listed following the notation "Included"; if the **-x** option is used, excluded deltas are listed following the notation "Excluded."

Identification Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing identification keywords with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the **m** flag in the file (see **admin**), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).

%E% Date newest applied delta was created (YY/MM/DD).

%G% Date newest applied delta was created (MM/DD/YY).

%U% Time newest applied delta was created (HH:MM:SS).

%Y% Module type: value of the t flag in the SCCS file. (See **admin.**)

%F% SCCS file name.

%P% Fully qualified SCCS file name.

%Q% The value of the **q** flag in the file. (See **admin.**)

%C% Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is not intended to be used on every line to provide sequence numbers.

%Z% The 4-character string **@(#)** recognizable by **what.**

%W% A shorthand notation for constructing **what** strings for XENIX program files %W%=%Z%%M%<horizontal-tab>%I%.

%A% A shorthand notation for constructing **what** strings for non-XENIX program files %A%=%Z%%Y%%M%%I%%Z%.

Files

Several auxiliary files may be created by **get.** These files are known generically as the g-file, l-file, p-file, and z-file. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name. The last component of all SCCS file names must be of the form **s.***module-name*; the auxiliary files are named by replacing the leading **s** with the tag. The g-file is an exception to this scheme: it is named by removing the **s.** prefix. For example, with **s.xyz.c,** the auxiliary file names would be **xyz.c, l.xyz.c, p.xyz.c,** and **z.xyz.c** respectively.

The g-file, which contains the generated text, is created in the current directory (unless the **-p** option is used). A g-file is created in all cases, whether or not any lines of text were generated by the **get.** It is owned by the real user. If the **-k** option is used or implied, its mode is 0644; otherwise, its mode is 0444. Only the real user need have write permission in the current directory.

The l-file contains a table showing which deltas were applied in generating the retrieved text. The l-file is created in the current directory if the **-l** option is used; its mode is 0444, and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the l-file have the following format:

- A blank character if the delta was applied; * otherwise
- A blank character if the delta wasn't applied or was applied and ignored; * if the delta wasn't applied and wasn't ignored
- A code indicating a "special" reason why the delta was or was not applied:

  "I": Included
  "X": Excluded
  "C": Cut off (by a **-c** option)

- Blank
- SCCS identification (SID)
- Tab character
- Date and time (in the form YY/MM/DD HH:MM:SS) of creation
- Blank
- Login name of person who created **delta**

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The p-file is used to pass information resulting from a **get** with an **-e** option along to **delta.** Its contents are also used to prevent a subsequent execution of **get** with an **-e** option for the same SID until **delta** is executed or the joint edit flag, **j,** (see **admin**) is set in the SCCS file. The p-file is created in the directory containing the SCCS file, and the effective user must have write permission in that directory. Its mode is 0644, and it is owned by the effective user. The format of the p-file is: the retrieved SID, a blank, the SID that the new delta will have when it is made, a blank, the login name of the real user, a blank, the date-time the **get** was executed, a blank and the **-i** option argument if it was present, a blank and the **-x** option argument if it was present, and a newline. There can be an arbitrary number of lines in the p-file at any time; no two lines can have the same new delta SID.

The z-file serves as a "lock-out" mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command that created it, namely **get.** The z-file is created in the directory containing the SCCS file for the duration of **get.** The same protection restrictions that apply to the **p-file** apply to the z-file. The z-file is created in mode 0444.

See Also

admin, delta, help, prs

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

what in "Commands" in the *XENIX 286 Reference Manual*

Diagnostics

Use **help** for explanations.

Notes

If the effective user has explicit or implicit write permission in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the **-e** option is used.

gets - Gets a string from the standard input.

Syntax

gets [*string*]

Description

gets can be used with **csh** to read a string from the standard input. If *string* is given, it is used as a default value should an error occur. The resulting string (either *string* or the string as read from the standard input) is written to the standard output. If no **default** is given and an error occurs, **gets** exits with exit status 1.

See Also

csh

line in "Commands" in the *XENIX 286 Reference Manual*

hdr - Displays selected parts of object files.

Syntax

hdr [-dhprSt] *file* ...

Description

hdr displays object file headers, symbol tables, and text or data relocation records in human-readable formats. It also prints out seek positions for the various segments in the object file.

a.out, x.out, and **x.out** segmented formats and archives are understood.

The symbol table format consists of six fields. In **a.out** formats, the third field is missing. The first field is the symbol's index or position in the symbol table, printed in decimal. The index of the first entry is zero. The second field is the type, printed in hexadecimal. The third field is the **s_seg** field, printed in hexadecimal. The fourth field is the symbol's value in hexadecimal. The fifth field is a single character that represents the symbol's type, as in **nm** except that **C** common is not recognized as a special case of "undefined." The last field is the symbol name.

If long form relocation is present, the format consists of six fields. The first is the descriptor, printed in hexadecimal. The second is the symbol ID, or index, in decimal. This field is used for external relocations as an index into the symbol table. It should reference an undefined symbol table entry. The third field is the position, or offset, within the current segment at which relocation is to take place; it is printed in hexadecimal. The fourth field is the name of the segment referenced in the relocation: **text, data, bss,** or **EXT** for external. The fifth field is the size of relocation: byte, word (2 bytes), or long (4 bytes). The last field, if present, will indicate that the relocation is relative.

If short form relocation is present, the format consist of three fields. The first field is the relocation command in hexadecimal. The second field contains the name of the segment referenced: **text** or **data.** The last field indicates the size of relocation: word or long.

Options are as follows:

-h Causes the object file header and extended header to be printed out. Each field in the header or extended header is labeled. This is the default option.

-d Causes the data relocation records to be printed out.

-t Causes the text relocation records to be printed out.

-r Causes both text and data relocation to be printed.

-p Causes seek positions to be printed out as defined by macros in the include file, **<a.out.h>**.

-s Prints the symbol table.

-S Prints the file segment table with a header (only applicable to files in segmented format).

See Also

nm

a.out in "File Formats" in the *XENIX 286 C Library Guide*

help - Asks for help about SCCS commands.

Syntax

help [*arg*] ...

Description

help finds information to explain a message from an SCCS command or to explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, **help** will prompt for one.

The arguments may be either message numbers, which normally appear in parentheses following messages, or command names of one of the following types:

type 1 Begins with nonnumerics, ends in numerics. The nonnumeric prefix is usually an abbreviation for the program or set of routines that produced the message (for example, **ge6** for message 6 from the **get** command).

type 2 Does not contain numerics (as a command, such as **get**)

type 3 Is all numeric (for example, **212**)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try **help stuck.**

Files

/usr/lib/help Directory containing files of message text

ld - Invokes the link editor.

Syntax

ld [*option*] ... *filename* ...

Description

ld is the XENIX link editor. It creates an executable program by combining one or more object files and copying the executable result into the file **a.out**. The *filename* must name an object or library file. These names must have the **.o** (for object) or **.a** (for archive) extensions. If one or more names are given, the names must be separated by one or more spaces. If errors occur while linking, **ld** displays an error message. The resulting **a.out** file is unexecutable.

ld concatenates the contents if the given object files in the order given in the command line. Library files in the command line are examined only if there are unresolved external references encountered from previous object files. Library files must be in **ranlib** format; that is, the first member must be named **__.SYMDEF**, which is a dictionary for the library. The library is searched iteratively to satisfy as many references as possible. Only those routines that define unresolved external references are concatenated. Object and library files are processed at the point they are encountered in the argument list, so the order of files in the command line is important. In general, all object files should be given before the library files. **ld** sets the entry point of the resulting program to be the beginning of the first routine.

ld has these options:

-F*num*
: Sets the size of the program to *num* bytes. Default stack size is 2 Kbytes.

-i Creates separate instruction and data spaces for small model programs. When the output file is executed, the program text and data areas are allocated separate physical segments. The text portion will be read-only and will be shared by all users executing the file.

-Ms Creates small model programs and checks for errors such as fixup overflow. This option is reserved for object files compiled or assembled using the small model configuration. This is the default model if no **-M** option is given.

-Mm Creates middle model programs and checks for errors. This option is reserved for object files compiled or assembled using the middle model configuration. This option implies **-i**.

-Ml Creates a large model program and checks for errors. The option is reserved for object files compiled using the large model configuration. This option implies **-i**.

-o *name*
Sets the executable program file name to *name* instead of **a.out.**

ld should be invoked by using the **cc** command instead of invoking it directly. **cc** invokes **ld** as the last step of compilation, providing all the necessary C language support routines. Invoking **ld** directly is not recommended since failure to give the command line arguments in the right order can result in errors.

Files

bin/ld

See Also

as, ar, cc, ranlib

Notes

The user must make sure that the most recent library versions have been processed with **ranlib** before linking. If this is not done, **ld** cannot create executable programs using these libraries.

lex - Generates programs for lexical analysis.

Syntax

lex [-ctvn] [*file*] ...

Description

lex generates programs to be used in simple lexical analysis of text. A file **lex.yy.c** is generated which, when loaded with the **lex** library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed.

The input *file* contains strings and expressions to be searched for, and C text to be executed when strings are found. Multiple files are treated as a single file. If no files are specified, standard input is used.

The options must appear before any files. The options are as follows:

-c Indicates C actions and is the default.

-t Causes the **lex.yy.c** program to be written instead to standard output.

-v Provides a one-line summary of machine-generated statistics.

-n Suppresses the - summary.

Strings and Operators

lex strings may contain square brackets to indicate character classes, as in **[abx-z]** to indicate a, b, x, y, and z; and the operators *, +, and ? mean, respectively, any nonnegative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. Thus, **[a-zA-Z]+** matches a string of letters. The character . is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r{d,e}* in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than | but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character $ at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in **yytext**, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols preceded by \.

Routines and Variables

Matching is done in order of the strings in the file. The actual string matched is left in **yytext,** an external character array. three subroutines defined as macros are expected: **input()** to read a character; **unput(c)** to replace a character read; and **output(c)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex(),** and the library contains a **main(),** which calls it. The action **REJECT** on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same **yytext;** and the function **yyless(p)** pushes back the portion of the string matched beginning at **p,** which should be between **yytext** and **yytext + yyleng.** The macros **input** and **output** use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout,** respectively. The external names generated by **lex** all begin with the prefix **yy** or **YY.**

lex File Format

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %% as in **YACC.** Lines that precede %% and begin with a nonblank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with braces ({}). Note that braces do not imply parentheses; only string substitution is done.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2000)

%n *n* number of states is *n* (500)

%t *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **-v** option, unless the **-n** option is used.

Example

```
 D          [0-9]
 %%
 if         printf("IF statement\n");
[a-z]+      printf("tag, value %s\n",yytext);
0{D}+       printf("octal number %s\n",yytext);
{D}+              printf("decimal number %s\n",yytext);
"++"              printf("unary op\n");
"+"               printf("binary op\n");
"/*"              { loop:
                  while (input() != \(fm\(**\(fm);
                  switch (input())
                          {
                          case \(fm/\(fm:   break;
                          case \(fm\(**\(fm:   unput(\(fm\(**\(fm);
                          default: go to loop;
                          }
                  }
```

See Also

yacc

Notes

This program translates its input into C source code, which in segmented programming environments is suitable for compiling as a small model program only. (See **cc.**)

lint - Checks C language usage and syntax.

Syntax

lint [-abchlnpuvx] *file* ...

Description

lint attempts to detect features of the C program *file* that are likely to be bugs or to be nonportable or wasteful. It also checks type usage more strictly than the compilers. Among the items currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

If more than one *file* is given, it is assumed that all files are to be loaded together; they are checked for mutual compatibility. If routines from the standard library are called from *file*, **lint** checks the function definitions using the standard lint library **llibc.ln.** If **lint** is invoked with the **-p** option, it checks function definitions from the portable lint library **llibport.ln.**

Any number of **lint** options may be used and in any order. The following options suppress certain kinds of complaints:

-a Suppresses complaints about assignments of long values to variables that are not long.

-b Suppresses complaints about **break** statements that cannot be reached. (Programs produced by **lex** or **yacc** will often result in a large number of such complaints.)

-c Suppresses complaints about casts that have questionable portability.

-h Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

-u Suppresses complaints about functions and external variables that are used and not defined or that are defined and not used. (This option is suitable for running **lint** on a subset of files of a larger program.)

-v Suppresses complaints about unused arguments in functions.

-x Does not report variables referred to by external declarations but never used.

The following arguments alter **lint**'s behavior:

-n Does not check compatibility against either the standard or the portable lint library.

-p Attempts to check portability to other dialects of C.

llibname
Checks functions definitions in the specified lint library. For example, **-lm** causes the library **llibm.ln** to be checked.

The **-D**, **-U**, and **-I** options of **cc** are also recognized as separate arguments.

Certain conventional comments in the C source will change the behavior of **lint:**

/*NOTREACHED*/
At appropriate points stops comments about unreachable code.

/*VARARGS*n*/
Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
Turns on the **-v** option for the next function.

/*LINTLIBRARY*/
Shuts off complaints about unused functions in this file.

lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

Files

Program Files

/usr/lib/lint[12]
/usr/lib/llibc.ln
/usr/lib/llibport.ln
/usr/lib/llibm.ln
/usr/lib/llibdbm.ln
/usr/lib/llibtermlib.ln

Standard Lint Libraries (Binary Format)

/usr/lib/llibc
/usr/lib/llibport
/usr/lib/llibm
/usr/lib/llibdbm
usr/lib/llibtermlib

See Also

cc

Notes

exit (see "System Functions" in the *XENIX 286 C Library Guide*) and other functions that do not return are not understood. This can cause improper error messages.

lorder - Finds ordering relation for an object library.

Syntax

lorder *file* ...

Description

The input is one or more object or library archive files. (See **ar.**) The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by **tsort** to find an ordering of a library suitable for one-pass access by **ld.**

Example

The following command intends to build a new library from existing **.o** files:

```
ar cr library "lorder *.o | tsort"
```

Files

***symref, *symdef**	Temporary files

See Also

ar, ld, tsort

Notes

Object files with names that do not end with **.o,** even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

m4 - Invokes a macro processor.

Syntax

m4 [options] [*files*] ...

Description

m4 is a macro processor intended as a front end for RATFOR, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is -, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

-e Operates interactively. Interrupts are ignored and the output is unbuffered.

-s Enables line sync output for the C preprocessor (#line ...).

-B*int*
Changes the size of the push-back and argument collection buffers from the default of 4,096.

-H*int*
Changes the size of the symbol table hash array from the default of 199. The size should be prime.

-S*int*
Changes the size of the call stack from the default of 100 slots. Macros take three slots, and nonmacro arguments take one.

-T*int*
Changes the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any **-D** or **-U** flags:

-D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

-U*name*
Undefines *name*.

Macro Calls

Macro calls have the form

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

The **(** must immediately follow the name of the macro. If a defined macro name is not followed by a **(**, it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. When the first character is not a digit, the macro name could consist of a letter, digit, or underscore (_).

Left and right single quotation marks are used to quote strings. The value of a quoted string is the string stripped of the quotation marks.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

m4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define
The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of **$***n* in the replacement of text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; **$#** is replaced by the number of arguments; **$*** is replaced by a list of all the arguments separated by commas; **$@** is like **$***, but each argument is quoted (with the current quotation marks).

undefine
Removes the definition of the macro named in its argument.

defn
Returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef
Like **define**, but saves any previous definition.

popdef
Removes current definition of its argument(s), exposing the previous ones if any.

ifdef
If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word XENIX is predefined in **m4**.

shift

Returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote

Changes quotation marks to the first and second arguments. The symbols may be up to five characters long. **changequote** without arguments restores the original values, namely left and right single quotation marks.

changecom

Changes left and right comment markers from the default **#** and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument, and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert

m4 maintains 10 output streams numbered 0 through 9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The **divert** macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert

Causes immediate output of text from diversions named as arguments, or all diversions if no argument exists. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum

Returns the value of the current output stream.

dnl

Reads and discards characters up to and including the next newline.

ifelse

Has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6, and 7. Otherwise, the value is either the fourth string or null if a fourth string is not present.

incr

Returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr

Returns the value of its argument decremented by 1.

eval

Evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include **+**, **-**, *****, **/**, **%**, **^** (exponentiation), bitwise **&**, **|**, **^**, and **~**; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len
Returns the number of characters in its argument.

index
Returns the position in its first argument where the second argument begins (zero-origin), or -1 if the second argument does not occur.

substr
Returns a substring of its first argument. The second argument is a zero-origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit
Transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include
Returns the contents of the file named in the argument.

sinclude
Identical to **include,** except that it says nothing if the file is inaccessible.

syscmd
Executes the XENIX command given in the first argument. No value is returned.

sysval
Is the return code from the last call to **syscmd.**

maketemp
Fills in a string of XXXXX in its argument with the current process ID.

m4exit
Causes immediate exit from **m4.** Argument 1, if given, is the exit code; the default is 0.

m4wrap
Argument 1 will be pushed back at final EOF; for example, **m4wrap(`cleanup()').**

errprint
Prints its argument on the diagnostic output file.

dumpdef
Prints current names and definitions, for the named items, or for all if no arguments are given.

traceon
With no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

traceoff
Turns off trace globally and for any macros specified. Macros specifically traced by **traceon** can be untraced only by specific calls to **traceoff.**

make - Maintains, updates, and regenerates groups of programs.

Syntax

make [-f *makefile*] [-p] [-i] [-k] [-s] [-r] [-n] [-b] [-e] [-t] [-q] [-d]
[*name*] ...

Description

The following is a brief description of all options and some special names:

-f *makefile*
Description file name. *makefile* is assumed to be the name of a description file. A file name of - denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

-p Prints out the complete set of macro definitions and target descriptions.

-i Ignores error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

-k Abandons work on the current entry, but continues on other branches that do not depend on that entry.

-s Silent mode. Does not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

-r Does not use the built-in rules.

-n No execute mode. Prints commands, but does not execute them. Even lines beginning with an **@** are printed.

-b Compatibility mode for old makefiles.

-e Environment variables override assignments within makefiles.

-t Touches the target files (causing them to be up-to-date) instead of issuing the usual commands.

-q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up-to-date.

-d Debug mode. Prints out detailed information on files and times examined.

.DEFAULT
If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS
Dependents of this target will not be removed when QUIT or INTERRUPT keys are pressed.

.SILENT
Same effect as the **-s** option.

.IGNORE
Same effect as the **-i** option.

make executes commands in *makefile* to update one or more target names. *name* is typically a program. If no **-f** option is present, **makefile, Makefile, s.makefile,** and **s.Makefile** are tried in order. If *makefile* is **-**, the standard input is taken. More than one **-f** *makefile* argument pair may appear.

make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, nonnull list of targets, then a colon (**:**), then a (possibly null) list of prerequisite files or dependencies. Text following a **;** and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or **#** begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><newline> sequence. A pound sign (**#**) and newline surround comments.

The following makefile says that **pgm** depends on two files **a.o** and **b.o,** and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h:**

```
pgm: a.o b.o
        cc a.o b.o -o pgm
a.o: incl.h a.c
        cc -c a.c
b.o: incl.h b.c
        cc -c b.c
```

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the **-s** option is present or the entry **.SILENT:** is in *makefile*, or unless the first character of the command is **@.** The **-n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed. (For more information, see discussion of the **MAKEFLAGS** macro in the "Environment" section, which follows.) The **-t** (touch) option updates the modified date of a file without executing any commands.

Commands returning nonzero status normally terminate **make.** If the **-i** option is present, or the entry **.IGNORE:** appears in *makefile*, or if the line specifying the command begins with <tab><hyphen>, the error is ignored. If the **-k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **-b** option allows old makefiles (those written for the old version of **make**) to run without errors. The difference between the old version of **make** and this version is that this version requires all dependency lines to have a (possibly null) command associated with them. The previous version of **make** assumed if no command was specified, the command was null.

INTERRUPT and QUIT cause the target to be deleted unless the target depends on the special name **.PRECIOUS.**

Environment

The environment is read by **make.** All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The **-e** option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by **make** as containing any legal input option (except **-f**, **-p**, and **-d**) defined for the command line. Further, upon invocation, **make** "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "supermakes". In fact, as noted above, when the **-n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make -n** recursively on a whole software system to see what would have been executed. This is because the **-n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE).** This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. Subsequent appearances of **$(***string1***[:***subst1***=[***subst2***]])** are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional **:***subst1***=***subst2* is a substitute sequence. If it is specified, all nonoverlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence appears in the "Libraries" section later in this entry.

Internal Macros

Five internally maintained macros are useful in writing rules for building targets:

$*** The macro **$* stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@** The **$@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<** The **$< macro is evaluated only for inference rules or the **.DEFAULT** rule. It is the module that is out of date with respect to the target (the "manufactured" dependent file name). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. Here is an example of making optimized **.o** files from **.c** files:

```
.c.o:
        cc -c -O $*.c
```

or:

```
.c.o:
        cc -c -O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target--essentially, those modules that must be rebuilt.

$% The **$%** macro is evaluated only when the target is an archive library member of the form **lib(file.o)**. In this case, **$@** evaluates to **lib** and **$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an uppercase D or F is appended to any of the four macros, the meaning is changed to directory part for D and file part for F. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, then **./** is generated. The only macro excluded from this alternative form is **$?**.

Suffixes

Certain names (for instance, those ending with .o) have default dependents such as .c and .s. If no update commands for such a file appear in **makefile** and if a default dependent exists, that prerequisite is compiled to make the target. In this case, **make** has inference rules that allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

The internal rules for **make** are contained in the source file **rules.c** for the **make** program. These rules can be locally modified. To print out the rules compiled into **make** on any machine and in a form suitable for recompilation, use the following command:

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the "null" string that **printf** (see "System Functions" in the *XENIX 286 C Library Guide*) prints when handed a null string.

A tilde in the above rules refers to an SCCS file. (See **sccsfile** in "File Formats" in the *XENIX 286 C Library Guide*.) Thus, the rule **.c~.o** would transform an SCCS C source file into the object file **.o**. Because the **s.** of the SCCS files is a prefix, it is incompatible with **make**'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (**.c:**) is the definition of how to build **x** from **x.c**. In effect, the other suffix is null. This is useful for building targets from only one source file (for example, shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

The default list is

```
.SUFFIXES: .o .c .y .l .s
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
	cc a.o b.o -o pgm
a.o b.o: incl.h
```

This is because **make** has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS, LFLAGS,** and **YFLAGS** are used for compiler options to **cc, lex,** and **yacc,** respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not as a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus **lib(*file*.o)** and **$(LIB)(*file*.o)** both refer to an archive library which contains *file*.o. (This assumes the **LIB** macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form **.*XX*.a** where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(*file*.o)** depend upon *file*.o explicitly. The most common use of the archive interface follows. This example assumes the source files are all C-type source:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

In fact, the **.c.a** rule is built into **make** and is unnecessary in this example. A more interesting but more limited example of an archive library maintenance construction follows:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?   @echo lib is now up to date
c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside **lib**) whose C source files are out-of-date. The substitution mode translates the **.o** to **.c.** (One cannot as yet transform **.o** to **.c~.**) Note also the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably, but it becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

Files

[Mm]akefile
s.[Mm]akefile

See Also

sh in "Commands" in the *XENIX 286 Reference Manual*

Notes

Some commands return nonzero status inappropriately; use **-i** to overcome this difficulty. Commands directly executed by the shell, notably **cd** (see "Commands" in the *XENIX 286 Reference Manual*) are ineffectual across newlines in **make.** The syntax **(lib(***file1***.o** *file2***.o** *file3***.o)** is illegal. You cannot build **lib(***file***.o)** from *file***.o.** The macro **$(a:.o=.c~)** is not available.

mkstr - Creates an error message file from C source.

Syntax

mkstr [-] *messagefile prefix file ...*

Description

mkstr is used to create files of error messages. Its use can decrease the size of programs with large numbers of error diagnostics. It can also reduce system overhead in running the program as error messages do not have to be constantly swapped in and out.

mkstr will process each specified *file*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. The optional dash (-) causes the error messages to be replaced at the end of the specified message file for recompiling part of a **mkstr**ed program.

A typical **mkstr** command line is

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file **pistrings** and processed copies of the source for these files to be placed in files whose names are prefixed with **xx**.

To process the error messages in the source to the message file, **mkstr** keys on the string **error("** in the input stream. Each time it occurs, the C string starting at the " is placed in the message file and followed by a null character and a newline character; the null character terminates the message so it can be easily used when retrieved, and the newline character makes it possible to sensibly **cat** the error message file to see its contents. The massaged copy of the input file then contains an **lseek** pointer into the file. This pointer can be used to retrieve the message, as the following example demonstrates. The command changes

```
error("Error on reading", a2, a3, a4);
```

into

```
error(m, a2, a3, a4);
```

where **m** is the seek position of the string in the resulting error message file. The programmer must create a routine **error** that opens the message file, reads the string, and prints it out. The "Example" section, which follows, illustrates such a routine.

Example

```
char    efilname [] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil<0) {
                efil = open(efilname, 0);
                if (efil < 0) {
                        perror(efilname);
                        exit(C);
                }
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

See Also

xstr

lseek in "System Functions" in the *XENIX 286 C Library Guide*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

All the arguments except the name of the file to be processed are unnecessary.

nm - Prints name list.

Syntax

nm [-acgnoOprsSuv] [+ *offset*] [*file*] ...

Description

nm prints the name list (symbol table) of each object file in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no **file** is given, the symbols in **a.out** are listed.

Each symbol name is preceded by its value in hexadecimal (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), **C** (common symbol), or **K** (8086/286 common segment). If the symbol table is in segmented format, symbol values are displayed as *segment*:*offset*. If the symbol is local (nonexternal), the type letter is in lowercase. The output is sorted alphabetically.

Options are

-a Print only absolute symbols.

-c Print only C program symbols (symbols that begin with "_") as they appeared in the C program.

-g Print only global (external) symbols.

-n Sort numerically rather than alphabetically.

-o Prepend file or archive element name to each output line rather than only once.

-O Print symbol values in octal.

-p Don't sort; print in symbol-table order.

-r Sort in reverse order.

-s Print 8086/286-executable file symbols as *segment*:*offset*.

-S Sort by size of symbol and display each symbol's size instead of value. The last symbol in each text or data segment may be assigned a size of 0. This option implies the **-i** and **-n** options.

-u Print only undefined symbols.

-v Also describe the object file and symbol table format.

For each symbol that is associated with an 8086/286-relocatable segment, this flag also causes the segment number to be printed after the symbol name, in the form of *s***01.**

This flag also causes a table of 8086/286 relocatable segments to be printed at the end of the name list. The segment table contains: the sequential segment number, the segment name, the class name, the type of required alignment, and the manner in which the segment can be combined with other segments.

Files

a.out Default input file

See Also

ar

ar, a.out in "File Formats" in the *XENIX 286 C Library Guide*

prof - Displays profile data.

Syntax

prof [-a] [-l] [*-low* [*-high*]] [*file*]

Description

prof interprets the file **mon.out** produced by the **monitor** subroutine. Under default modes, the symbol table in the named object file (**a.out** default) is read and correlated with the **mon.out** profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **-a** option is used, all symbols (rather than just external symbols) are reported. If the **-l** option is used, the output is listed by symbol value rather than by decreasing percentage.

To cause calls to a routine to be tallied, the **-p** option of **cc** must have been given when the file containing the routine was compiled. This option also arranges for the **mon.out** file to be produced automatically.

Files

mon.out	For profile
a.out	For namelist

See Also

cc

monitor, profil in "System Functions" in the *XENIX 286 C Library Guide*

Notes

Beware of quantization errors.

If you use an explicit call to **monitor,** you will need to make sure that the buffer size is equal to or smaller than the program size.

prs - Prints an SCCS file.

Syntax

prs [-d[*dataspec*]] [-r[*SID*]] [-e] [-l] [-a] file ...

Description

prs prints, on the standard output, all or part of an SCCS file (see **sccsfile** in "File Formats" in the *XENIX 286 C Library Guide*) in a user-supplied format.

If a directory is named, **prs** behaves as though each file in the directory were specified as a named file, except that non-SCCS files (the last component of whose path names does not begin with **s.**) and unreadable files are silently ignored.

If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to **prs**, which may appear in any order, consist of options and file names. All the described options apply independently to each named file:

-d[*dataspec***]** Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file data keywords (see "Data Keywords," which follows) interspersed with optional user-supplied text.

-r[*SID***]** Used to specify the SCCS identification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

-e Requests information for all deltas created earlier than and including the delta designated via the **-r** option.

-l Requests information for all deltas created later than and including the delta designated via the **-r** option.

-a Requests printing of information for both removed deltas (delta type = **R**) and existing deltas (delta type = **D**). See **rmdel** for more information. If the **-a** option is not specified, information for existing deltas only is provided.

Data Keywords

Data keywords specify the parts of an SCCS file that are to be retrieved and output. All parts of an SCCS file have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by **prs** consists of the user-supplied text and appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either **simple**, in which keyword substitution is direct, or **multiline**, in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords. A tab is specified by **\t** and carriage return/newline is specified by **\n**.

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	"
:Li:	Lines inserted by Delta	"	nnnnn	"
:Ld:	Lines deleted by Delta	"	"	"
:Lu:	Lines unchanged by Delta	"	"	"
:DT:	Delta type	"	**D** or **R**	"
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	"
:R:	Release number	"	nnnn	"
:L:	Level number	"	"	"
:B:	Branch number	"	"	"
:S:	Sequence number	"	"	"
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	"
:Dy:	Year Delta created	"	nn	"
:Dm:	Month Delta created	"	"	"
:Dd:	Day Delta created	"	"	"
:T:	Time Delta created	"	:Th:::Tm:::Ts:	"
:Th:	Hour Delta created	"	nn	"
:Tm:	Minutes Delta created	"	"	"
:Ts:	Seconds Delta created	"	"	"
:P:	Programmer who created Delta	"	logname	"
:DS:	Delta sequence number	"	nnnn	"
:DP:	Predecessor Delta seq-no.	"	"	"
:DI:	Seq-no. of deltas incl, excl, ignored	"	:Dn:/:Dx:/:Dg:	"
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	"
:Dx:	Deltas excluded (seq #)	"	"	"
:Dg:	Deltas ignored (seq #)	"	"	"
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	"	"
:UN:	User names	User Names	"	"
:FL:	Flag list	Flags	"	"
:Y:	Module type flag	"	"	S
:MF:	MR validation flag	"	**yes** or **no**	"
:MP:	MR validation pgm name	"	text	"
:KF:	Keyword error/warning flag	"	**yes** or **no**	"

:BF:	Branch flag	"	"	"
:J:	Joint edit flag	"	"	"
:LK:	Locked releases	"	:R:...	"
:Q:	User defined keyword	"	text	"
:M:	Module name	"	"	"
:FB:	Floor boundary	"	:R:	"
:CB:	Ceiling boundary	"	"	"
:Ds:	Default SID	"	:I:	"
:ND:	Null Delta flag	"	**yes** or **no**	"
:FD:	File descriptive text	Comments	text	M
:BD:	Body text	Body	"	"
:GB:	Gotten body	"	"	"
:W:	A form of **what** string	N/A	:Z::M:\t:I:	S
:A:	A form of **what** string	N/A	:Z::Y: :M: :I::Z:	"
:Z:	**what** string delimiter	N/A	@(#)	"
:F:	SCCS file name	N/A	text	"
:PN:	SCCS file path name	N/A	"	"

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

Examples

The following command

```
prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce this on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

The following command

```
prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r
s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a **special case**

```
prs s.file
```

may produce on the standard output

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only option argument allowed to be used with the *special case* is the **-a** option.

Files

/tmp/pr?????

See Also

admin, delta, get, help

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Refer to **help** for explanations.

ranlib - Converts archives to random libraries.

Syntax

ranlib *archive*

Description

By adding a table of contents named **__.SYMDEF** to the beginning of the *archive*, **ranlib** converts each *archive* to a form that the loader can load more rapidly. **ranlib** uses **ar** to reconstruct the archive so that sufficient temporary file space must be available in the file system containing the current directory.

See Also

ld, ar

copy, settime in "Commands" in the *XENIX 286 Reference Manual*

Notes

Failure to process or reprocess a library with **ranlib** will cause **ld** to fail. Because generation by **ar** and randomization by **ranlib** are separate, phase errors are possible. The loader **ld** warns when the modification date of a library is more recent than the creation of its dictionary; but this means you receive a warning even if you only copy the library.

ratfor - Converts Rational FORTRAN into standard FORTRAN.

Syntax

ratfor [*option* ...] [*filename* ...]

Description

ratfor converts a rational dialect of FORTRAN into ordinary FORTRAN. **ratfor** provides control flow constructs essentially identical to those in C:

statement grouping:

```
{ statement; statement; statement }
```

decision making:

```
if (condition) statement [ else statement ]
switch (integer value) {
        case integer:  statement
        ...
        [ default: ] statement
}
```

loops:

```
while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [ until (condition) ]
break [n]
next [n]
```

and some additional syntax to make programs easier to read and write:

Free form input:

multiple statements/line; automatic continuation

Comments:

```
# this is a comment
```

Translation of relationals:

>, > = , etc., become .GT., .GE., etc.

Return (*expression*):

returns expression to caller from function

Define:

define *name replacement*

Include:

include *filename*

The option **-h** causes quoted strings to be turned into 27H constructs. **-C** copies comments to the output, and attempts to format it neatly. Normally, continuation lines are marked with an **&** in column 1; the option **-6x** makes the continuation character **x** and places it in column 6.

Notes

This program translates its input into C source code, which in segmented programming environments, is suitable for compiling as a small model program only. (For more information, see **cc.**)

regcmp - Compiles regular expressions.

Syntax

regcmp [-] *file*

Description

regcmp, in most cases, precludes the need for calling **regcmp** from C programs. This saves on both execution time and program size. The command **regcmp** compiles the regular expressions in *file* and places the output in *file*.**i.** If the - option is used, the output will be placed in *file*.**c.** The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotation marks.

The output of **regcmp** is C source code. Compiled regular expressions are represented as **extern char** vectors. *file*.**i** files may thus be included into C programs, or *file*.**c** files may be compiled and later loaded. In the C program that uses the **regcmp** output, **regex(abc,line),** applies the regular expression named **abc** to **line.** Diagnostics are self-explanatory.

Examples

```
name    "([A-Za-z][A-Za-z0-9_]*)$0"

telno   "\({0,1}([2-9][01][1-9])$0\){0,1} *"
        "([2-9][0-9]{2})$1[ -]{0,1}"
        "([0-9]{4})$2"
```

In the C program that uses the **regcmp** output

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named **telno** to **line.**

See Also

regex in "System Functions" in the *XENIX 286 C Library Guide*

rmdel - Removes a delta from an SCCS file.

Syntax

rmdel -r*SID file* ...

Description

rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must not be that of a version being edited for the purpose of making a delta. That is, if a **p-file** (see **get**) exists for the named SCCS file, the SID specified must not appear in any entry of the **p-file.**

If a directory is named, **rmdel** behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of whose path names does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; nonSCCS files and unreadable files are silently ignored.

Files

x-file See **delta**

z-file See **delta**

See Also

delta, get, help, prs

sccsfile in "File Formats" in the *XENIX 286 C Library Guide*

Diagnostics

Use **help** for explanations.

sact - Prints current SCCS file editing activity.

Syntax

sact *file* ...

Description

sact informs the user of any impending deltas to a named SCCS file. This situation occurs when **get** with the **-e** option has been executed without a subsequent execution of **delta.** If a directory is named on the command line, **sact** behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of - is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1 Specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta

Field 2 Specifies the SID for the new delta to be created

Field 3 Contains the logname of the user who will make the delta (executed a **get** for editing)

Field 4 Contains the date that **get -e** was executed

Field 5 Contains the time that **get -e** was executed

See Also

delta, get, unget

Diagnostics

Use **help** for explanations.

sccsdiff - Compares two versions of an SCCS file.

Syntax

sccsdiff -r*SID1* -r*SID2* [-p] [-s*n*] *file* ...

Description

sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

-r*SID?* *SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to **bdiff** in the order given.

-p Pipe output for each file through **pr.**

-s*n* *n* is the file segment size that **bdiff** will pass to **diff.** This is useful when **diff** fails due to a high system load.

Files

/tmp/get????? Temporary files

See Also

get, help

bdiff, pr in "Commands" in the *XENIX 286 Reference Manual*

Diagnostics

file: No differences if the two versions are the same

Use **help** for explanations.

size - Prints the size of an object file.

Syntax

size [*object* ...]

Description

size prints the (decimal) number of bytes required by the text, data, and bss portions and their sum in octal and decimal for each object-file argument. If no file is specified, **a.out** is used.

See Also

a.out in "File Formats" in the *XENIX 286 C Library Guide*

spline - Interpolates smooth curve.

Syntax

spline [*option*] ...

Description

spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output has two continuous derivatives and enough points to look smooth when plotted.

The following options are recognized, each as a separate argument.

-a Supplies abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

-k The constant **k** used in the boundary value computation

$$y''_0 = ky'_1, \ldots, y''_n = ky'_{n-1}$$

is set by the next argument. By default **k =0.**

-n Spaces output points so that approximately *n* intervals occur between the lower and upper **x** limits. (Default *n* = **100.**)

-p Makes output periodic, i.e., matches derivatives at ends. First and last input values should normally agree.

-x Next 1 (or 2) arguments are lower (and upper) **x** limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

Diagnostics

When data is not strictly monotone in **x**, **spline** reproduces the input without interpolating extra points.

Notes

A limit of 1000 input points is silently enforced.

stackuse - Determines stack requirements for C programs.

Syntax

stackuse [-nl*maxidlen*] [-m*startsym*] [-e*linkage*]
[-r*fakeref*] [-s*libstack*] [-l*library*] [-v] [-o*name*]
[*file* ...]

Description

stackuse determines the stack requirements of one or more C language programs. It displays the name of the **main** routine in a file, its stack requirements in words, and the number of recursive routines. All command line switches are optional.

-**nl***maxidlen* On output, uses alternate maximum identifier length. If zero is specified, an internal maximum is used.

-**m***startsym* Uses an alternate start ('main') symbol. If *startsym* is 0, the first symbol encountered is used.

-**e***linkage* Uses a different cost for linkage on entry to a routine. Linkage includes saved registers, return address.

-**r***fakeref* Uses the named file *fakeref* as a fake references file. The format is: *parent child*. The special *parent*.**LEAF** is a meta-parent meaning all leaf nodes.

-**s***libstack* Uses the named file as library of costs for external routines. The format is: *subr stack*. The special *subr*.**UNDEF** is a meta-subroutine meaning all undefined routines.

-**l***library* Uses a system-provided *libstack* for standard libraries, for example, -**lc**, -**ll**, -**ly**.

-**v** Prints verbose output. Creates **calltree**, **selfcost**, **cost**, other files. Names are **stack.ct**, **stack.slf**, **stack.tot**. To change the name, use the -**o** option.

-**o***name* Uses different *name* for output files. Files are named *name*.**ct**, *name*.**slf**, *name*.**tot**.

The -**r** and -**s** options may be repeated an arbitrary number of times. The effect is additive rather than destructive. In the case of duplicate definitions, the first is used.

Lines of the -**r** and -**s** files which begin with a pound sign (**#**) are treated as comments and are ignored otherwise.

Diagnostics

Usage (fatal).

Redefinitions in **-r**, **-s** files, or in the source (warning).

Calltree with multiple roots (warning).

Presence of recursive routine (warning).

Presence of routines for which no stack value is provided (warning).

Files

/usr/lib/stackuse/*	Passes, libraries
/tmp/*	Temporaries used by passes.

Notes

The scanner can be fooled by complex C constructs.

For the *libstack* and *fakeref* files, a comment character (**#**) is used.

strings - Finds the printable strings in an object file.

Syntax

strings [-] [-o] [-*number*] *file* ...

Description

strings looks for ASCII strings in a binary file. A string is any sequence of four or more printing characters ending with a newline or a null character. Unless the - flag is given, **strings** only looks in the initialized data space of object files. If the **-o** flag is given, then each string is preceded by its decimal offset in the file. If the *-number* flag is given then *number* is used as the minimum string length rather than 4.

strings is useful for identifying random object files and many other items.

See Also

hd, od in "Commands" in the *XENIX 286 Reference Manual*

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

strip - Removes symbols and relocation bits.

Syntax

strip [dehrsStx] *file* ...

Description

strip removes selected parts of an object file, including the header, text, data, relocation records, and symbol table. **strip** works directly on the named files; nothing is written to the standard output.

strip is typically used to remove symbol table and relocation information from a file after debugging has been completed. It also is useful for creating a compact namelist file in which text and data have been removed.

-d Strip data and the data relocation records.

-e Strip the extended header.

-h Strip the header and extended header.

-r Strip all relocation records except the **x.out** short form.

-s Strip the symbol table.

-S Strip the segment table.

-t Strip text and the text relocation records.

-x Strip all relocation records.

strip has the same effect as the **-s** option of **ld**. If no options are given, the **-r** and **-s** options are implied.

Although **strip** can be used to remove an **x.out** header from an 8086/286-relocatable file, it cannot be used to remove run-time relocation records.

Files

/tmp/stm* Temporary file

See Also

a.out in "File Formats" in the *XENIX 286 C Library Guide*

ld in "Commands" in the *XENIX 286 Reference Manual*

time - Times a command.

Syntax

time *command*

Description

The given *command* is executed; after it is complete, **time** prints the elapsed time during the command ("real"), the time spent in the kernel ("sys"), and the time spent in execution of the command outside of the kernel ("user"). Times are reported in seconds.

The times are printed on the standard error output.

See Also

times in "System Functions" in the *XENIX 286 C Library Guide*

tsort - Sorts a file topologically.

Syntax

tsort [*file*]

Description

tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

See Also

lorder

Diagnostics

Odd data: There is an odd number of fields in the input file.

Notes

The sorting algorithm is quadratic, which can be slow if you have a large input list.

unget - Undoes a previous **get** of an SCCS file.

Syntax

unget [-r*SID*] [-s] [-n] *file* ...

Description

unget undoes the effect of a **get -e** done prior to creating the intended new delta. If a directory is named, **unget** behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of - is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Options apply independently to each named file.

-r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by **get** as the "new delta.") The use of this option is necessary only if two or more versions of the same SCCS file have been retrieved for editing by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

-s Suppresses the printout, on the standard output, of the intended delta's SID.

-n Causes the retention of the file that would normally be removed from the current directory.

See Also

delta, get, sact

Diagnostics

Use **help** for explanations.

val - Validates an SCCS file.

Syntax

val -

val [-s] [-r*SID*] [-m*name*] [-y*type*] *file* ...

Description

val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to **val** may appear in any order. The arguments consist of options, which begin with a -, and named files.

val has a special argument, -, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

val generates diagnostic messages on the standard output for each command line and file processed. It also returns a single 8-bit code upon exit.

The options are defined as follows. The effects of any options apply independently to each named file on the command line:

-s The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

-r*SID* The argument value *SID* (SCCS identification string) is an SCCS delta number. A check is made to determine if the SID is ambiguous (for example, **r1** is ambiguous because it physically does not exist but implies **1.1**, **1.2**, etc., which may exist) or invalid (**r1.0** or **r1.1.0** is invalid because neither case can exist as a valid delta number). If the SID is valid and not ambiguous, a check is made to determine if it actually exists.

-m*name* The argument value *name* is compared with the SCCS **%M%** keyword in *file*.

-y*type* The argument value *type* is compared with the SCCS **%Y%** keyword in *file*.

The 8-bit code returned by **val** is a disjunction of the possible errors. That is, it can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = Missing file argument

bit 1 = Unknown or duplicate keyletter argument

bit 2 = Corrupted SCCS file

bit 3 = Can't open file or file not SCCS

bit 4 = *SID* is invalid or ambiguous

bit 5 = *SID* does not exist

bit 6 = **%Y%, -y** mismatch

bit 7 = **%M%, -m** mismatch

Note that **val** can process two or more files on a given command line and in turn can process multiple command line (when reading the standard input). In these cases, an aggregate code is returned: a logical OR of the codes generated for each command line and file processed.

See Also

admin, delta, get, prs

Diagnostics

Use **help** for explanations.

Notes

val can process up to 50 files on a single command line.

xref - Cross-references C programs.

Syntax

xref [*file*] ...

Description

xref reads each named *file* or the standard input if no file is specified and prints a cross reference consisting of lines of the form

identifier filename line_number ...

Function definition is indicated by a plus sign (+) preceding the line number.

See Also

cref

xstr - Extracts strings from C programs.

Syntax

xstr [-c] [-] [*file*]

Description

xstr maintains a file **strings** into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings and is most useful if they are also read-only strings.

The command

xstr -c *name*

will extract the strings from the C source in *name*, replacing string references by expressions of the form **(&xstr[number])** for some number. An appropriate declaration of **xstr** is prepended to the file. The resulting C text is placed in the file **x.c**, to then be compiled. The strings from this file are placed in the **strings** data base if they are not there already. Repeated strings and strings that are suffixes of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file **xs.c** declaring the common **xstr** space can be created by a command of the form

xstr -c *name1 name2 name3 . . .*

This **xs.c** file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared), saving space and swap overhead.

xstr can also be used on a single file. A command

xstr *name*

creates files **x.c** and **xs.c** as before, without using or affecting any **strings** file in the same directory.

It may be useful to run **xstr** after the C preprocessor if any macro definitions yield strings or if there is conditional code containing strings that may not, in fact, be needed. **xstr** reads from its standard input when the argument - is given. An appropriate command sequence for running **xstr** after the C preprocessor is

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

xstr does not touch the file **strings** unless new items are added. Thus, **make** can avoid remaking **xs.o** unless truly necessary.

Files

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array **xstr**
/tmp/xs*	Temp file when **xstr** *name* doesn't touch **strings**

See Also

mkstr

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

If a string is a suffix of another string in the data base, but the shorter string is seen first by **xstr**, both strings will be placed in the data base even when placing just the longer one there will do.

yacc - Invokes a compiler-compiler.

Syntax

yacc [-vd] *grammar*

Description

yacc converts a context-free grammar into a set of tables for a simple automaton that executes an LR(1)-parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program **yyparse**. This program must be loaded with the lexical analyzer program, **yylex**, as well as **main** and **yyerror**, an error handling routine. These routines must be supplied by the user; **lex** is useful for creating lexical analyzers usable by **yacc**.

If the **-v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **-d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the **yacc**-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

Files

y.output

y.tab.c

y.tab.h	Defines for token names
yacc.tmp, yacc.acts	Temporary files
/usr/lib/yaccpar	Parser prototype for C programs

See Also

lex

Diagnostics

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

Notes

Because file names are fixed, one **yacc** process at most can be active in a given directory at a time.

intel®

APPENDIX C
RELATED PUBLICATIONS

Copies of the following publications can be ordered from

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Overview of the XENIX 286 Operating System, Order Number 174385 -- XENIX history, XENIX uses, basic XENIX concepts, and an overview of other XENIX manuals.

XENIX 286 Installation and Configuration Guide, Order Number 174386 -- how to install XENIX on your hardware and tailor the XENIX configuration to your needs.

XENIX 286 User's Guide, Order Number 174387 -- a tutorial on the most-used parts of XENIX, including terminal conventions, the file system, the screen editor, and the shell.

XENIX 286 Visual Shell User's Guide, Order Number 174388 -- a XENIX command interface ("shell") that replaces the standard command syntax with a menu-driven command interpreter.

XENIX 286 System Administrator's Guide, Order Number 174389 -- how to perform system administrator chores such as adding and removing users, backing up file systems, and troubleshooting system problems.

XENIX 286 Reference Manual, Order Number 174390 -- all commands in the XENIX 286 Basic System.

XENIX 286 Programmer's Guide, Order Number 174391 -- (this manual) XENIX 286 Extended System commands used for developing and maintaining programs.

XENIX 286 C Library Guide, Order Number 174542 -- standard subroutines used in programming with XENIX 286, including all system calls.

XENIX 286 Device Driver Guide, Order Number 174393 -- how to write device drivers for XENIX 286 and add them to your system.

XENIX 286 Text Formatting Guide, Order Number 174541 -- XENIX 286 Extended System commands used for text formatting.

XENIX 286 Communications Guide, Order Number 174461 -- installing, using, and administering XENIX networking software.

C is described in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. One copy is supplied with Intel's XENIX product. Additional copies can be ordered from the publisher, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

intel®

COMMAND INDEX

adb, 6-1, B-3
admin, B-12
ar, B-17
as, 7-1, B-19

cb, B-21
cc, 2-1, B-22
cdc, B-27
comb, B-30
config, B-32
cref, B-36
csh, 8-1, B-38
ctags, B-58

delta, B-60

get, B-63
gets, B-69

hdr, B-70
help, B-72

ld, B-73
lex, 9-1, B-75
lint, 3-1, B-78
lorder, B-81

m4, 11-1, B-82
make, 4-1, B-86
mkstr, B-93

nm, B-95

prof, B-97
prs, B-98

ranlib, B-102
ratfor, B-103
regcmp, B-105
rmdel, B-106

sact, B-107
SCCS, 5-1, B-12, B-27, B-30, B-60, B-63, B-72, B-98, B-106, B-107, B-108, B-118,B-119
sccsdiff, B-108
size, B-109
spline, B-110
stackuse, B-111
strings, B-113
strip, B-114

time, B-116
tsort, B-117

unget, B-118

val, B-119

xref, B-121
xstr, B-122

yacc, 10-1, B-124

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does this publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating) _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, Oregon 97123

ISO-N TECHNICAL PUBLICATIONS HF2-1-830

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

SOFTWARE

0470/7K/0685/WCP/AD